THE UNIVERSITY
of EDINBURGH

Lightweight Speculative Support for Aggressive Auto-Parallelisation Tools

Daniel C. Powell

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2015

Abstract

With the recent move to multi-core architectures it has become important to create the means to exploit the performance made available to us by these architectures. Unfortunately parallel programming is often a difficult and time-intensive process, even to expert programmers. Auto-parallelisation tools have aimed to fill the performance gap this has created, but static analysis commonly employed by such tools are unable to provide the performance improvements required due to lack of information at compile-time. More recent aggressive parallelisation tools use profiled-execution to discover new parallel opportunities, but these tools are inherently unsafe. They require either manual confirmation that their changes are safe, completely ruling out auto-parallelisation, or they rely upon speculative execution such as software thread-level speculation (SW-TLS) to confirm safe execution at runtime.

SW-TLS schemes are currently very heavyweight and often fail to provide speedups for a program. Performance gains are dependent upon suitable parallel opportunities, correct selection and configuration, and appropriate execution platforms. Little research has been completed into the automated implemention of SW-TLS programs.

This thesis presents an automated, machine-learning based technique to select and configure suitable speculation schemes when appropriate. This is performed by extracting metrics from potential parallel opportunities and using them to determine if a loop is suitable for speculative execution and if so, which speculation policy should be used. An extensive evaluation of this technique is presented, verifying that SW-TLS configuration can indeed be automated and provide reliable performance gains. This work has shown that on an 8-core machine, up to 7.75× and a geometric mean of 1.64× speedups can be obtained through automatic configuration, providing on average 74% of the speedup obtainable through manual configuration.

Beyond automated configuration, this thesis explores the idea that many SW-TLS schemes focus too heavily on recovery from detecting a dependence violation. Doing so often results in worse than sequential performance for many real-world applications, therefore this work hypothesises that for many highly-likely parallel candidates, discovered through aggressive parallelisation techniques, would benefit from a simple dependence check without the ability to roll back. Dependence violations become extremely expensive in this scenario, however this would be incredibly rare. With a thorough evaluation of the technique this thesis confirms the hypothesis whilst achiev-

ing speedups of up to 22.53×, and a geometric mean of 2.16× on a 32-core machine. In a competitive scheduling scenario performance loss can be restricted to at least sequential speeds, even when a dependence has been detected.

As a means to lower costs further this thesis explores other platforms to aid in the execution of speculative error checking. Introduced is the use of a GPU to offload some of the costs to during execution that confirms that using an auxiliary device is a legitimate means to obtain further speedup. Evaluation demonstrates that doing so can achieve up to 14.74× and a geometric mean of 1.99× speedup on a 12-core hyperthreaded machine. Compared to standard CPU-only techniques this performs slightly slower with a geometric mean of 0.96× speedup, however this is likely to improve with upcoming GPU designs.

With the knowledge that GPU's can be used to reduce speculation costs, this thesis also investigates their use to speculatively improve execution times also. Presented is a novel SW-TLS scheme that targets GPU-based execution for use with aggressive auto-parallelisers. This scheme is executed using a competitive scheduling model, ensuring performance is no lower than sequential execution, whilst being able to provide speedups of up to 99× and on average 3.2× over sequential. On average this technique outperformed static analysis alone by a factor of 7× and achieved approximately 99% of the speedup obtained from manual parallel implementations and outperformed the state-of-the-art in GPU SW-TLS by a factor of 1.45.

Lay Summary

Recent trends in computer design have more towards using multiple processors inside a single computer. These processors have the ability to perform several tasks at once, in parallel, allowing for an overall increase in the number of tasks they are able to complete in a set amount of time. Taking advantage of this is, however, a complex and time intensive process, even for experienced programmers. The problem is that when you perform multiple tasks at once, each task may interfere with each other, potentially causing errors to occur.

Instead of having programmers create tasks that don't conflict, there has been a lot of research into automatically converting tasks such that they can be performed in parallel, however these methods are not very good at doing so. Some automated methods are more aggressive and instead predict when tasks can run in parallel without conflicting, performing much better than the non-aggressive methods, however their predictions are not always correct. When they are not correct errors can occur. A new way of running tasks in parallel is to take the tasks suggested by the aggressive means and run them in parallel anyway, instead keeping track of what every task is doing. This is called running them speculatively. If two tasks interfere with each other, they are stopped, the changes they have made a undone and the tasks are allowed to continue one by one, ensuring that they don't interact with each other. These methods work relatively well, but when tasks interact they often end up taking longer overall to be performed than if they were simply run one after the other in the first place. Also, sometimes the checking to ensure that they do not interact also takes longer.

This thesis investigates ways to automatically determine whether it will be faster to run multiple tasks speculatively, or to just run them one after another. If it will be faster to run them speculatively then this thesis also presents a method to automatically determine which way to ensure two tasks haven't interacted will be the fastest.

Further to that this thesis hypothesises that some aggressive methods are very good at detecting which tasks can be ran safely in parallel, and instead of using a speculation technique that focuses on very quick recovery if they do interact, instead focuses on running each task as fast as possible, possibly with the aid of extra processing devices. This becomes extremely costly when two tasks do interact, effectively

meaning they have to be restarted entirely, but that is extremely rare.

Finally there are different types of processors in existence, standard style processors found in every computer, and customised ones that used to be used solely for displaying items on a computer screen. These customised processors have been found to be very quick at performing hundreds of tasks at once, but only in very specific circumstances. They also suffer from the same problems as standard processors in that tasks could interact. This thesis investigates ways to use both the aggressive techniques to find tasks that can run in parallel, and the speculative techniques to ensure that errors don't occur when tasks do interact. This is a technique that has had wide investigation on standard processors, but very little investigation on these customized processors.

Acknowledgements

Many thanks to my supervisor and my girlfriend, for the constant support they have provided; my tireless proofreading elves; and of course, my kettle for not breaking down when I needed it the most.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Daniel C. Powell*)

Publications

The following refereed conference papers (in reverse chronological order) have been published during the course of this PhD. These form the basis for parts of this thesis as indicated.

- **Daniel C. Powell**, Björn Franke. "Safety Net: Lightweight Software-TLS Support for Probably Parallel Applications." Currently under peer review.
  — Chapter 6 is partially based on this paper.

- Zheng Wang, **Daniel C. Powell**, Björn Franke, Michael O'Boyle. "Exploitation of GPUs for the Parallelisation of Probably Parallel Legacy Code." In: *Proceedings of 23rd International Conference on Compiler Construction (CC'14), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'14)*, Grenoble, France, April 2014.
  — Chapter 7 is based on this paper.

- **Daniel C. Powell**, Björn Franke. "An Integrated Approach to Software Thread-Level Speculation: Machine-Learning Based Policy and Parameter Selection." In: *Proceedings of HIPEAC Compiler, Architecture and Tools Conference (CATC'12)*, Haifa, Israel, November 2012.
  — Chapters 4 and 5 are based on this paper.

Table of Contents

1 Introduction **1**

1.1 Modern Parallel Architectures 2
1.2 Automated Parallelisation 3
1.3 Speculative Parallelisation 4
1.3.1 Hardware or Software 5
1.4 Motivation 6
1.5 Goals 6
1.6 Hypotheses 7
1.7 Structure 7

2 Background: Thread Level Speculation **9**

2.1 Parallelism and Dependences 10
2.1.1 DOALL loops 10
2.1.2 DOWHILE loops 11
2.1.3 Thread Blocks 12
2.1.4 Transactional Memory 13
2.1.5 Dependence Types 14
2.1.5.1 Safe Data Dependences 14
2.1.5.2 False Data Dependences 15
2.1.6 Other Hazards 15
2.2 Automated Parallelism Discovery 16
2.2.1 Static Analysis 16
2.2.2 Execution Profiling 17
2.2.3 Performance Considerations 18
2.3 Speculative Execution 18
2.3.1 Speculation Workflow 19

2.3.2 When to Check . . . 22
2.3.2.1 Lazy Checking . . . 22
2.3.2.2 Eager Checking . . . 24
2.3.2.3 Combined Checking . . . 24
2.3.3 Trace Topologies . . . 25
2.3.3.1 Distributed Traces . . . 25
2.3.3.2 Centralised Traces . . . 26
2.3.3.3 Hybrid Topologies . . . 27
2.3.4 Synchronisation . . . 27
2.3.4.1 Barriers . . . 28
2.3.4.2 Master Thread . . . 28
2.3.4.3 Overlaps . . . 29
2.3.5 Trace Structures and Collision Detection . . . 29
2.3.5.1 Bitsets . . . 30
2.3.5.2 Counters . . . 31
2.3.5.3 Address Lists . . . 32
2.3.6 Accuracy and False Dependences . . . 32
2.3.6.1 Addressing Size . . . 33
2.3.6.2 Hashing Techniques . . . 33
2.3.6.3 Single or Multiple Traces . . . 34
2.3.7 Version Management . . . 35
2.3.7.1 Management Scheme . . . 35
2.3.7.2 Versioning Schedule . . . 36
2.3.7.3 Data Structures and Topology . . . 37
2.3.7.4 Value Forwarding . . . 38
2.3.7.5 Granularity . . . 38
2.4 Conclusions . . . 39

3 Related Work **41**
3.1 Profile-Driven Parallelism Detection . . . 41
3.2 CPU Schemes . . . 43
3.2.1 S-TLS . . . 43
3.2.2 POLYLIBTLS . . . 44
3.2.2.1 SPLSC . . . 44
3.2.2.2 SPLIP . . . 45

3.2.3 STMLite . . . 47
3.2.4 DSWP & SMTX . . . 47
3.3 GPGPU Speculation . . . 49
3.3.1 PARAGON . . . 49
3.4 Hardware Based Speculation . . . 51
3.4.1 Transactional Synchronisation Extensions . . . 51
3.5 Conclusion . . . 52

4 Lightweight Pipelined Speculation 55
4.1 Speculative Storage Structure . . . 56
4.2 Pipeline Stages and Execution Workflow . . . 57
4.3 Supported Dependences . . . 58
4.4 Conflict Detection . . . 58
4.5 Empirical Evaluation . . . 60
4.5.1 Experimental Methodology . . . 60
4.5.2 Summary of Key Results . . . 61
4.6 Conclusion . . . 63

5 Smart Speculation Policy Selection 65
5.1 Motivating Example . . . 65
5.2 SW-TLS Configuration . . . 67
5.2.1 Factors Affecting Performance . . . 68
5.2.2 Common Speculation Parameters . . . 68
5.3 Policy Selection Workflow . . . 69
5.3.1 Prediction Model Training . . . 69
5.3.2 Policy Calculation . . . 70
5.4 Empirical Evaluation . . . 71
5.4.1 Evaluation Methodology . . . 71
5.4.2 Policy Selection Testing . . . 71
5.4.2.1 Machine Learning Techniques . . . 72
5.5 Summary of Key Results . . . 73
5.6 Conclusions . . . 78

6 Automated Error Checking for Aggressive Parallelisation 81
6.1 Parallelisation Target . . . 82
6.2 Memory Trace Data Structure . . . 83

6.2.1 Page Caching 85
6.2.2 Structure Usage 86
6.2.2.1 Allocation/Initialisation 86
6.2.2.2 Trace 87
6.3 Simple Distributed Error Detection 88
6.3.1 Scalability 90
6.4 Reduction-Tree Error Detection 90
6.4.1 Scalability 95
6.5 GPU Conflict Detection 95
6.6 Automated Program Transformations 101
6.7 Empirical Evaluation 104
6.7.1 Auto-Parallelisation Analysis 106
6.7.2 Page Table Statistics 108
6.7.3 Simple Distributed Detection Scheme 112
6.7.4 Reduction-Tree Detection Scheme 114
6.7.4.1 Comparison to Simple Distributed Scheme 114
6.7.5 Hybrid CPU-GPU Detection Scheme 116
6.7.6 Dependence Violations 118
6.8 Conclusion 119

7 GPU-Based Speculation **121**
7.1 Motivation 122
7.2 Execution Workflow 124
7.3 Speculative GPU Execution 125
7.3.1 Speculative Data Structures 125
7.3.2 Violation Detection 126
7.3.2.1 Speculative Load 127
7.3.2.2 Speculative Store 127
7.3.2.3 Flow Dependence 128
7.3.2.4 Anti Dependence 128
7.3.2.5 Output Dependence 128
7.3.3 Comparison to Other Approaches 128
7.4 Compilation and Code Transformations 129
7.4.1 Parallelism Detection 129
7.4.1.1 Speculative Variables 130

7.4.2 OpenCL Code Generation . . . 130
7.4.3 Code Merging . . . 131
7.5 Experimental Setup . . . 131
7.5.1 Platform . . . 131
7.5.2 Benchmarks . . . 131
7.5.3 Compiler and Evaluation Runs . . . 133
7.5.4 Comparison . . . 133
7.6 Empirical Evaluation . . . 134
7.6.1 Overall Results . . . 134
7.6.2 Comparison with the Statically Safe Approach . . . 135
7.6.3 Comparison with Paragon . . . 136
7.6.4 Comparison to Manually Parallelized Code . . . 137
7.6.5 Analysis . . . 138
7.6.5.1 Limitation of Static Analysis . . . 138
7.6.5.2 Speculation Costs . . . 139
7.6.5.2.1 Dependence Violation . . . 140
7.7 Conclusion . . . 140

8 Conclusion **143**
8.1 Contributions . . . 144
8.1.1 Automated Policy Selection . . . 144
8.1.2 Lightweight Error Checking . . . 144
8.1.3 GPU-Based Speculative Execution . . . 145
8.1.4 Pipelined Speculation Scheme . . . 146
8.2 Analysis and Future Work . . . 146
8.2.1 Limitations of Policy Selection . . . 146
8.2.2 Scalable Centralised Error Detection . . . 147
8.2.3 Block Tracing . . . 147
8.2.4 Combination CPU-GPU Speculation . . . 148

Bibliography **149**

Chapter 1

Introduction

Since the dawn of the computing era the demands on computing power have been ever increasing. As time has progressed faster processors have been developed to meet those demands, combined with newer tools, languages and techniques to support that power. In an effort to better utilise the resources that are available many different execution paradigms have been developed in many forms most of which are loosely based around some form of parallel processing. This ranges from the very high level such as shared mainframes in the 50s, allowing many distinct users to work in parallel by sharing processing resources split into timeslices, to a lower level such as allowing multiple processes to execute on a machine at once through scheduling, to the very low level such as pipelining instructions so that different parts of each instruction are executing at the same time.

Each of these styles of execution have a very clear limitation, the single threaded-execution model [42]. For many years however that limitation has been overcome by relying upon the development of faster, more complex processing architectures to increase the performance of applications. This is such a well known and relied upon trend that it was developed into a law deciding the abilities of future architectures, Moore's Law [39]. Moore's Law surmised that over time the density of components that could fit on a single die would increase and the cost per component would continue to fall. In 1975 the rule became more formalised in that the circuit density would double roughly every 24-months [46]. The effects of this law is demonstrated in Figure 1.1. This density of components translated into a direct performance increase for processors with adaptations of the law stating that the performance of such processors

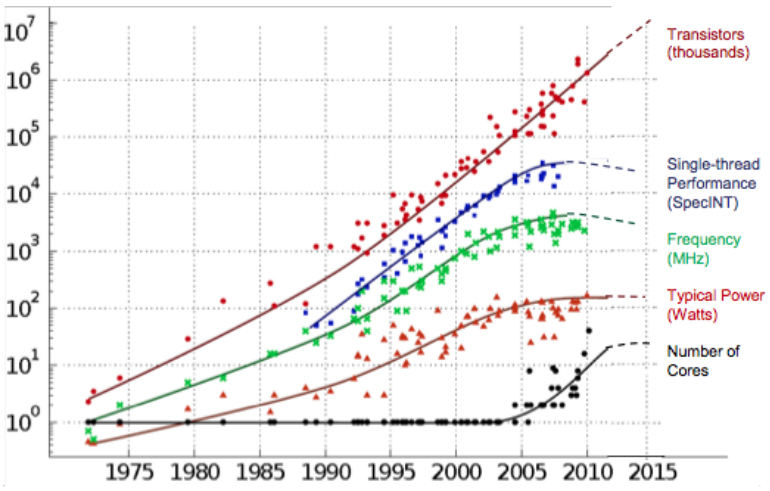


Figure 1.1: Processor trends for 1970-2012. The graph shows consistent exponential growth of the number of elements on a die (Moore's Law) and the introduction of multicore designs due to flattening of performance, frequency and power growth trends. Source: C.Moore [27]

would double approximately every 18 months, a trend that was able to be maintained into the mid-2000s. Beyond that however it became apparent that processor designers had started to reach the physical limits of a single-threaded processor causing exorbitant power consumption and thermal output requirements [5]. Ultimately this required a complete shift in architectural design, a move towards parallel processing and multicore designs.

1.1 Modern Parallel Architectures

Despite the long history of development that parallel processing and architectures have had [4] their widespread use has only taken off within the last decade. Before that single-threaded microprocessors were good enough and continued to provide performance increases year over year, demonstrated by Figure 1.1. Since that decline multicore architectures have found their way into virtually all modern computing devices ranging from mobile phones, games consoles, personal computers, laptops all the way to today's large-scale data centres and supercomputers. Producing parallel programs targetting these processors is one of today's grand challenges in computer systems research [38]. Despite the progress in parallel programming languages and systems [8], programming of multicore platforms still remains a skilled activity in the hands of a few expert programmers [17].

Further complexity exists due to the range of different architectural designs available ranging from the more common chip multiprocessors, to more heterogeneous designs with differently designed processing units for specific tasks. The designs of heterogeneous processors are wide in variety, for example the Cell [18] multiprocessor that has additional units designed for specific powerful computations, to devices with power consumption considerations, such as ARM's big.LITTLE processor, one core for slower low-power processing and another for faster, power intensive processing [20]. Rapidly becoming a viable programming platform is the use of general purpose GPUs [11] whose memory architecture and processing unit designs require extremely different programming considerations to obtain useful performance [6]. This adds further difficulty to well performing parallel programming and makes auto-parallelisation techniques a very desirable solution.

1.2 Automated Parallelisation

As with parallel architectures, automated parallelisation has undergone decades of research [22]. Initially these were based around the automated vectorisation of programs to speed up single-threaded programs that were then translated into parallelising techniques. More explicit parallel languages have been developed to aid in the auto-parallelisation process [8], however these languages require a large investment to learn a new programming paradigm. This approach also does not aid in the parallelisation of the vast quantities of legacy code that exist [23].

The applicability of much research into auto-parallelisers is often restricted to niche settings such as array-based numerical computations [41]. Outside these specialist domains the performance of even state-of-the-art auto-parallelisers on real-world codes is highly disappointing [44]. One of the key reasons for this dramatic failure is that compilers are notoriously weak at raising the abstraction level [31]. This includes inference of dependence patterns from sequential applications, where such information is not explicitly expressed by the programmer, yet it is essential for successful parallelisation. In fact, solutions to the general data dependence problem are not computable [7].

A large number of static dependence analyses, which compute approximations of the actual dependence information, have been developed [33]. These analyses are necessarily conservative, i.e. they may report the presence of a so-called *may* dependence

even if this dependence does not materialise for any of the legal program inputs, but no actual dependence will remain undetected. Unfortunately, most static analyses report *may* dependences overly frequently and this, in turn, prevents parallelisation. Such *may* dependences can also be triggered through the use of many common programming techniques, such as pointer aliasing and indirect addressing.

A recent approach to capturing dependence information more precisely is based on profiled execution [44]. For this technique a program is instrumented and then executed. The resulting profile is analysed and checked for dependences. Such profile-guided parallelisation schemes are effective at uncovering non-statically analysable parallel loops, however it lacks a correctness guarantees for all inputs different to the ones used during profiling allowing for undetected dependences to exist in the program, ultimately making it unsafe to execute without further manual analysis or run-time checking.

1.3 Speculative Parallelisation

Thread-level speculation (TLS) has attracted the attention of a number of researchers, e.g. [32, 9, 43], as a means of adding safety to potentially unsafely parallelised programs. This is performed by speculatively executing possibly independent work items such as loop iterations or function calls on multiple processor cores. Through the use of TLS a program is executed with additional memory-access tracking code. For each potentially unsafe region of code the memory accesses performed are recorded and then at relevant intervals they are analysed to detect any data dependences that may have occurred. Prior to executing any potentially unsafe regions of code a checkpoint is made which, if a dependence is detected the program rolls back to and re-executes safely.

Most TLS frameworks are geared towards the case when the majority of work items are independent and can be executed in parallel, but every so often dependence violations require work items in flight to be squashed. In fact, almost all work has focused on TLS for when there is a realistic chance of serialising dependences, i.e. discarding of non-committed memory writes and rollback to safe state is needed. This, for example, may happen in loops that are sometimes parallel and sometimes not, depending on the actual data inputs provided to the program.

1.3.1 Hardware or Software

To support TLS various hardware [19, 10] and software schemes [29, 28, 24, 25, 36, 30] have been developed for the provision and management of buffers and dependence detection mechanisms.

Hardware supported TLS (HW-TLS) includes the use of specifically tasked hardware-based buffers for memory tracing and recovery. These schemes are generally very reliable and provide significant speedups to a given program whilst being very accurate at detecting dependences that occur. The implementation of speculative execution of a program can also happen completely automatically in the hardware making it a very desirable feature. However, HW-TLS requires customised hardware to support its execution. Designing and implementing such hardware can be very costly and requires large regions of a CPU die to perform correctly. This makes the process very expensive for hardware designers when there exist easier targets for optimisation, that provide more demonstratable speedup and consume less of the available resources on chip. Hardware-based speculation also suffers from fixed limits on the size of memory buffers restricting the ability to speculate over large sections of code. As such, there are currently no full commercially available implementations of HW-TLS. Instead there is one architecture that supports a subset of TLS features, Intel's Haswell [15]. Haswell uses Transactional Synchronisation Extensions (TSX) to provide hardware transactional support that, with additional software support, could be extended to implement a full TLS system. Haswell's architecture is discussed further in Section 3.4.

In contrast, software-based TLS (SW-TLS) is much more flexible. The size of buffers used can be customised to the needs of a given program and the methods of storing traces, detecting dependences and even performing rollbacks/commits of speculative state can be changed to suit the needs of both the program and the platform it is executed on. SW-TLS is also very cheap to implement, requiring no customised hardware. SW-TLS can be executed on any multi-core machine. The specific designs of SW-TLS can also be explored cheaply requiring no simulators or additional hardware. However, SW-TLS is very unreliable at providing a respectable speedup and generally slower than its hardware equivalent. The flexibility that it provides results in performance varying based on the program that is being speculated on, the hardware used to execute the program and the specific type of speculation being performed. In many cases the use of SW-TLS can result in a slowdown of the original sequential program,

but, given the correct circumstances and configuration SW-TLS can provide significant speedups. As such it is essential that an appropriate SW-TLS configuration be selected for a given program.

1.4 Motivation

To further the state of auto-parallelising technologies it is desirable to add safety to some of the more aggressive auto-parallelisers that currently exist. TLS appears to be suitable for this but due to the costly nature and unavailability of HW-TLS it is impractical to consider it as an option and SW-TLS is currently too unreliable at providing speedup to truly be considered. No research has as yet been performed into the automated use of SW-TLS.

Through profiled execution it is clear that there are many missed opportunities when it comes to parallel execution. Profiling can also provide a high-confidence prediction that a loop is probably parallel and, as such, does not require some of the more heavyweight techniques such as those geared towards recovering from a dependence violation that many SW-TLS focus on. There are also additional platforms, such as GPUs that have not yet been widely targetted by SW-TLS methods.

1.5 Goals

The goal of this thesis is to extend the existing knowledge and technologies surrounding software-based thread-level speculation. It intends to do this by:

- Describing research into automating the speculative process by:
  - Creating a means to select which loops will profit from speculative execution,
  - Automating the process of inserting speculative parallel markup into existing sequential programs,
  - Developing a means for automatically selecting an appropriate speculation scheme, and tune it so that significant, reliable speedup can be obtained.

- Investigating the hypothesis that standard heavyweight SW-TLS methods are unnecessary by:
  - Creating a very lightweight speculation scheme that prioritises fast execution over rollback.
  - Attempting to offload part of the speculative process to an auxiliary processing device such as a GPU.
- Attempting to create a useful lightweight speculation scheme for use on a general purpose GPU.

1.6 Hypotheses

It is the hope that this thesis can prove that additional performance can be obtained completely automatically and safely through the use of code profiling and speculative execution. As such it is desirable to prove that SW-TLS can be configured completely automatically to provide a near-optimal solution that provides consistent performance increases over sequential code.

This thesis also argues that in many circumstances it can be predicted with enough confidence that a given, potentially unsafely parallelised region of code does not contain any data dependences, and as such can execute with very little support for safety and rollbacks. Instead it is theorised that any version control normally used by SW-TLS schemes can be discarded in favour of an extremely lightweight dependence detection scheme that merely confirms that no dependence has occurred.

Finally, it is hypothesised that the GPU is both an appropriate target for SW-TLS and that it can be a useful tool to provide further speedup of standard CPU based SW-TLS schemes.

1.7 Structure

The rest of this thesis is structured as follows:

- Chapter 2 provides a thorough investigation into the principles of parallel execution and speculative execution, with a detailed analysis of the benefits and disadvantages of each method.

- Chapter 3 provides a look at already existing auto-parallelisation schemes and SW-TLS schemes with an evaluation of how well they perform and their limitations.
- Chapter 4 provides an alternative speculation scheme that attempts to address some niche scenarios that existing schemes perform poorly at.
- Chapter 5 introduces an automated loop selection process that identifies loops that will benefit from speculative execution and selects a suitable speculation policy that will provide improved performance.
- Chapter 6 investigates the hypothesis that heavy-weight speculative schemes are often not necessary and instead programs can execute only with simple confirmation that dependence violations have not occurred. This chapter also investigates the use of a GPU as an auxiliary unit to offload some of the speculative processing to.
- Chapter 7 introduces a new, automated scheme to take sequential programs and execute them speculatively on a GPU.
- Chapter 8 summarises the final discoveries and conclusions of this work, along with a look at possible future work beyond this thesis..

Chapter 2

Background: Thread Level Speculation

As a way to progress beyond the end of Moore's law and continue increasing performance beyond the physical limitations of single-threaded execution, commercial processor architects have adopted the multi-core processor design as the de-facto standard in modern, general purpose computing systems. There is also a rapidly growing trend of using lower-power multi-core architectures in mobile platforms such as mobile phones and tablets, improving user interaction and increasing performance on these platforms. As such it is essential that the applications executed on these platforms are designed to exploit these parallel architectures and utilise their resources effectively. Manual parallelisation is a difficult, time-intensive task [23], and automatic parallelisation techniques using static analysis, while achieving some performance increases, have left much to be desired in more complex real-world applications [44]. Software thread-level speculation is intended to address these limitations.

Prior to addressing the designs of modern SW-TLS techniques, this chapter provides a brief background on frequently encountered types of parallel execution and the hazards that they can introduce in Section 2.1. This chapter then discusses static analysis and the automated parallelism discovery and mapping methods that this work targets in Section 2.2. An in-depth description of common speculation techniques is provided in Section 2.3 with final conclusions provided in Section 2.4.

```
1 for (int i = 0; i < n; i++) {
2   a[i] = b[i] + c[i];
3 }
```

Figure 2.1: A simple DOALL based loop that would benefit from parallelisation.

2.1 Parallelism and Dependences

Application-level parallelism is a diverse and complex subject with many different styles of implementation. Application-level parallelism also creates the possibility of various hazards that can occur causing the incorrect execution of a program. This section first lists several different parallel programming paradigms that commonly employ TLS, then goes on to list several hazards that TLS must account for.

2.1.1 DOALL loops

One of the simplest ways to implement parallelism is to split the processing of individual iterations of a standard DOALL loop between multiple threads [22]. Such loops include DOALL loops in Fortran, and `for` and `foreach` loops as implemented in C/C++/Java and many other imperative languages. An example loop that would benefit from parallelisation can be found in Figure 2.1. This style of parallelism can also be extended to more functional programming constructs such as list comprehensions.

To parallelise a DOALL loop it is essential to be able to calculate the number of iterations and the values of their loop condition variables, be it a simple counter such as in `for` loop, or a collection of objects as in `foreach` loops and list comprehensions. These values must be determinable prior to the start of execution so that each iteration can be distributed evenly among the available worker threads. Additionally these loops must not contain `break` statements as these can prevent some iterations from executing. If they do then additional support is required to prevent future iterations from executing. These conditions provide a guarantee that all iterations of the loop will execute allowing them to be distributed evenly between all worker threads. They also provide a guarantee that the loop execution condition does not contain any data dependences, however the iteration body of the loop may contain data dependences that must be handled appropriately.

```
1 while (has_next(item)) {
2   item = get_next(item);
3   ...
4 }
```

Figure 2.2: A simple DOWHILE based loop that could benefit from parallelisation.

In TLS these loops are the simplest to implement allowing for a flexible tracking and dependence detection scheme. The guarantee that all iterations will execute allows for dependence checks to be deferred until a point after the loop has executed, and the explicit order in which each iteration is executed allows for simple rollback methods ensuring a deterministic output.

2.1.2 DOWHILE loops

A more complex construct to parallelise is the DOWHILE loop. Traditional DOWHILE loops continue to execute each iteration whilst a certain condition remains true. This condition is normally updated within the iteration body itself. An example of a loop doing so can be seen in Figure 2.2. This makes it more difficult to determine how many iterations the loop will execute and hence makes it harder to parallelise by distributing iterations between each thread. Additional static or runtime analysis is required to determine how many iterations and the runtime state of each iteration before parallel execution can begin.

Many other loops can fit into this category such as DOALL loops that contain `break` statements, or DOALL loops whose run condition relies upon functions or on variables other than those in the initialisation stage.

In TLS these loops are complex to implement as they are likely to require speculation across their execution condition. For more eager parallelisation schemes one or more iterations may execute beyond the final run condition making it important to allow for the side effects of these iterations to be rolled back individually. This also restricts when dependence checks can be performed as they may have to occur at the end of every iteration instead of allowing them to be delayed until after the loop has finished. However, as with simple DOALL loops the order in which each iteration is executed is very strict ensuring a deterministic output.

```
1 for(int i = 0; i < n; i++) {
2   a += b[i];
3 }
4 for(int j = 0; j < n; j++) {
5   c += d[i];
6 }
```

Figure 2.3: A simple two-stage process that could be split into two thread blocks.

2.1.3 Thread Blocks

Beyond loops is the concept of threading, where each thread executes code based upon the specific task assigned to it. Whilst each thread is often contributing towards the same goal, each executes different instructions to achieve their assigned task, often working in tandem with the other threads through the use of barriers and synchronisation and communication between threads using shared memory or message passing. Figure 2.3 provides a simple example of a region of code that can be easily split into two threads. Many parallel programming paradigms have been devised to make use of this concept such as producer-consumer models [21], where one thread generates an object (the producer), which is then passed on to another thread which processes it and generates the required output (the consumer). A more complex version of this is pipelined processing, where each thread is assigned a particular stage of the processing with the result of each passed to the next thread to complete their stage. Costs associated with this form of processing are based on the setup and destroy times taken to fill each stage of the pipeline. However, once full the pipeline can process in parallel a number of items equal to its depth.

Alternately, threads can be working on completely separate tasks using shared resources, producing their own individual output. Examples of this are databases and web servers where each client request is performed in a separate thread. These threads are less likely to require direct communication, but may require locks and other synchronisation based on access to shared resources.

In TLS pipelined processing and parallel tasks are complex to handle. Pipelined processing ensures the order in which tasks are tasks completed but does not necessarily guarantee the consistent ordering of access to shared resources. Dependence checks

```
1 void addAmount(index, amount) {
2   entry = database.get(index);
3   entry += amount;
4   database.put(index, entry);
5 }
```

Figure 2.4: A simple database process that could use transactional support.

must be performed at any synchronisation points, but the type of synchronisation restricts how these checks can be performed, often with a thread having to compare its progress to every other executing thread before being allowed to continue. This makes checking more costly as each thread is unable to share the checking costs between the other threads as with loops. Similarly with parallel tasks there is no concept of ordering for each thread to access or update shared resources, forcing checks and rollbacks to be performed on a first-come-first-served basis.

2.1.4 Transactional Memory

A concept similar to TLS in thread blocks is often employed, for example, in database systems [45]. This is transactional memory where, on each access to the database, a thread will start a transaction. Figure 2.4 provides a simple database process that may benefit from the use of transactional memory. The thread will request the information it requires to continue, process them and generate the results to be committed back to the database. Once this has been done the transaction will attempt to commit itself, first checking if other threads have accessed the same resources. If they have, the process will have to be repeated until the thread is able to perform the required actions without interference from other threads or clients. This process can lead to hazards such as live-lock where the thread is constantly prevented from performing its required tasks as other threads block it from doing so. Additionally, as with thread blocks, there is no easily defined ordering for each transaction, again resulting in a first-come-first-served processing basis.

2.1.5 Dependence Types

During access to shared resources by parallel threads there are several similar but distinct data dependences that can occur, resulting in incorrect output. In each of these cases if the access was performed sequentially by a single thread an error would not occur and the resulting output would be deterministic and correct; however, as parallel threads can access data in any order each dependence can also be performed in any order, resulting in a non-deterministic program and potentially an error in the final output of a program.

Read-After-Write Dependences When one thread reads an element of shared data followed by another thread writing to that same element there is a *flow* or *read-after-write* (RAW) dependence on that element.

Write-After-Read Dependences When one thread writes an element of shared data followed by another thread reading that same element there is an *anti* or *write-after-read* (WAR) dependence on that element.

Write-After-Write Dependences When one thread writes an element of shared data followed by another thread also writing that same element there is an *output* or *write-after-write* (WAW) dependence on that element.

2.1.5.1 Safe Data Dependences

A *safe* data dependence is one that occurs during execution between iterations but the specific order in which it was executed does not cause an error in the final output. Specifically if the ordering in which the dependence is executed is the same as that of a sequential version of the program this would result in a *safe* data dependence. For instance, if a write by a programmatically earlier iteration by a parallel thread occurs before the read or write of a later iteration then there is a data dependence between the two iterations, but the order in which the dependence executed is the same as it would be if it were executed sequentially, hence it is a *safe* dependence. By not being able to distinguish between standard dependences and *safe* dependences, deferred detection would trigger an unnecessary rollback.

2.1.5.2 False Data Dependences

A *false* data dependence is one that is detected by speculative tracking schemes that did not actually occur in a program. This is frequently triggered by schemes that employ memory reduction methods such as memory address hashing or larger than byte size addressing. These methods are described in more detail in Section 2.3.6. *False* data dependences often trigger unnecessary rollbacks or other version control techniques resulting in additional processing and longer execution times, potentially much longer than the original sequential version of a program.

2.1.6 Other Hazards

By allowing code that potentially contains data dependences to execute can also trigger several other hazards. In particular these include:

Inifinite Loops

If the execution condition of a loop is the subject of a data dependence, or derived from any variable that may contain a data dependence could result in an inconsistently executed loop. This can include the execution of additional iteration, missing iterations and possibly even an inifinite loop. The simplest solution to this is to ensure the loop condition is not derived from an unsafe variable, however this a potentially difficult task to ensure automatically. Worse still, many speculation schemes may never detect this error as dependence checks occur after the loop has finished executing.

Segfaults

In cases of indirect memory addressing and pointer arithmetic it is possible for a memory access to be performed outside of its expected address range. For instance an access of an array may cross the boundaries of the array itself into the memory of other variables. This can be handled as the value causing the erroneous access would be detected as a dependence violation. A special case of this is to access memory outside of the program itself, triggering a segfault. This is easily recoverable by speculative execution by overriding the segfault handler and treating the fault as a failed speculation. An extreme example would be to override a loop varible resulting in an infinite loop, or an access to non-speculative memory that cannot be restored. These are difficult issues that have not been widely investigated by speculative research.

Irreversible I/O

Many programs perform I/O during their standard operation. Should any occur during speculative execution it may be impossible to reverse the consequences should a dependence be detected. This can be handled through buffering of both input and output during speculative regions.

2.2 Automated Parallelism Discovery

This section reviews several methods and their limitations for performing automated parallelism detection. The simplest form of parallel execution to detect and implement automatically is that of DOALL loops, as described in the previous section. Hence, most auto-parallelising compilers and tools focus heavily on these loops [14]. As this dissertation is intended to be an extension on top of existing auto-parallelising tools it will focus mainly on these same loops.

2.2.1 Static Analysis

The primary method for detecting parallelism in sequentially executing code is static analysis [14]. During this stage compilers/parallelising tools generate a model of a program and analyse the data dependences that may exist. The ordering of every read and write performed on an array/variable is analysed to extract whether or not a flow, anti or output dependence exists on that variable. The program is then split into basic blocks that can or cannot be parallelised and rewritten into separate threads based on various performance metrics embedded into the tool. A common case is the parallelisation of loops where static analysis is used to detect cross-iteration data dependences that would prevent the parallel execution of individual iterations.

In many circumstances sequential code may re-use a variable to calculate new data that is independent of prior uses of that variable, for instance a piece of data calculated during each iteration of a loop. In the sequential program this re-use would not cause an error, but during parallel execution this variable must be *privatised* such that each thread has its own copy, which may be used without interfering with other threads. Many existing parallelising tools will automatically detect these variables and privatise them correctly.

Similarly, in many circumstances a *reduction* on a variable will occur, for instance the summing of a value across iterations of a loop. Due to the inherent data dependence that exists on such a reduction variable, many compilers will determine this case as unparallelisable. However, such cases are simple to parallelise by creating a thread-private copy of the reduction variable and, at the end of parallel execution, adding additional code to perform the reduce operation across the privatised copies to the final output variable.

Dependence detection, privatisation and reduction are relatively simple exercises on individual variables, but each of these phases become significantly more difficult when programs involve the use of arrays or pointer manipulation, a very common circumstance. When a program uses arrays a tool must pre-calculate every index into the array, or identify a common access pattern to the array to determine whether no data dependences exist for each access of the array. Similarly a compiler must be able to determine which piece of data a pointer is accessing to be able to provide a guarantee that no data dependences exist. These are significant tasks for a parallelisation tool to perform, relying upon advanced knowledge of the program being executed that simply cannot be represented by many intermediate representations used during static analysis. The problem becomes even worse in circumstances of pointer aliasing or the use of indirect array indexing that prevent the compiler from determining any array access patterns. In these circumstances many parallelising tools and compilers are necessarily conservative and determine these sections of code to be unparallelisable to prevent possible errors in the program from occurring. However, in doing so they miss many possible opportunities for obtaining faster programs through parallel execution.

2.2.2 Execution Profiling

To extract further parallel opportunities from sequential code, static analysis can be augmented with profiled execution analysis [44, 48]. To perform profiled execution a sequential program is instrumented with additional code to allow it to generate traces of all memory operations that a program will execute. Such traces include pointers into the original high-level code that indicate which section is being executed: for instance the start and finish of each iteration of a loop. These logs containing additional program structure can then be analysed to determine if a data dependence occurred during execution.

This technique is meant to be used as an addition to static analysis. In many cases static analysis can prove with absolute certainty that no data dependence exists, and also in many cases that a dependence definitely exists. This technique is for use in circumstances where static analysis cannot prove either.

The analysis of memory trace logs can provide absolute certainty for cases where a dependence exists but where static analysis has failed. In cases where no dependence is detected profiling can provide no guarantee that a dependence does not exist due to alternative execution scenarios, such as different input data sets to the program. Many tools leave the final decision of whether a loop can be parallelised to the programmer using the tool, delegating the consequences of incorrectly identified parallelism to the programmer. In fully automated parallelisation tools this situation is untenable, partly because programmers themselves are fallible.

2.2.3 Performance Considerations

Modifying a program to execute in parallel can provide significant speedups, but the process also introduces additional overheads related to thread creation and management, thread communication and synchronisation. These overheads can easily outweigh any benefits that parallel execution may present. During parallelisation many tools include various metric analyses to determine if a speedup will be obtained. These metrics are not precise but are a useful indicator of the possible performance benefits of parallelisation. Similar tools exist for profiled execution, with the added benefit that performance can be measured during profiling. The possible speed estimated by these metrics can also vary depending on the configuration of the program, input data sets and the platform performing the execution. As yet, no such metrics exist for speculative execution.

2.3 Speculative Execution

This section provides an in-depth description of many common speculation techniques, with analysis of their suitability in various scenarios and their effects on the performance and resource utilisation of a program.

This section begins with a description of the standard workflow of S-TLS systems in

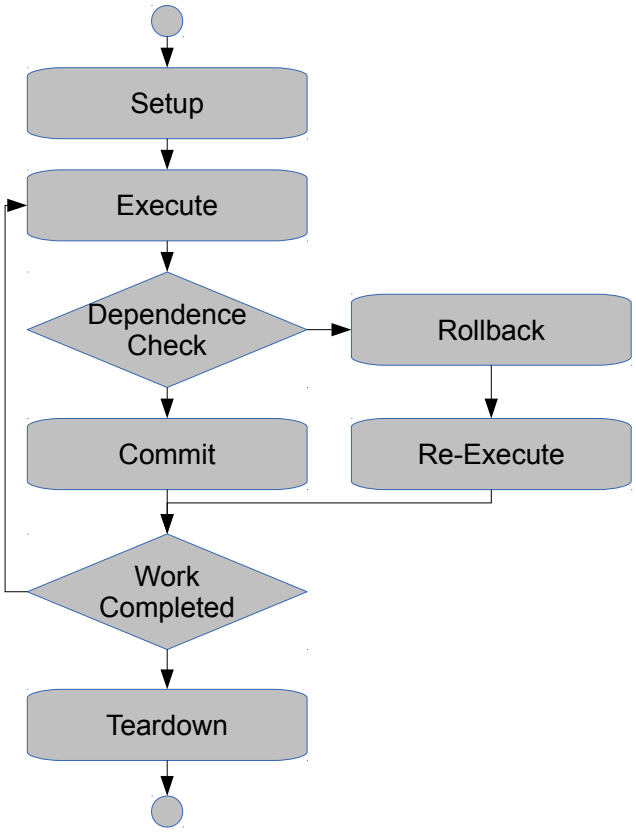


Figure 2.5: Generalised Workflow of Common Speculative Execution Schemes

Section 2.3.1, before analysing the effects on the ordering of that workflow in Section 2.3.2. A description of storage layouts used for speculative data is provided in Section 2.3.3 and then the various methods of synchronisation commonly used by S-TLS in Section 2.3.4. Next a description of the data structures used to store memory traces is given in Section 2.3.5 with an analysis of the techniques used to reduce the size of those structures and the hazards that can occur by doing so in Section 2.3.6. Finally a description of the version management methods used to provide the underlying safety of S-TLS is provided in Section 2.3.7.

2.3.1 Speculation Workflow

Thanks to the nature of speculative execution most schemes follow a similar pattern of execution. This general workflow can be seen in Figure 2.5. Execution of a program is split up into sequential, parallel and speculative sections based on the analysis performed on the program. When a speculative region of code is encountered the following stages are performed:

(i) **Set-up.** At the start of speculative execution, or at some point prior, the structures necessary to store the speculative state are created and reset. These structures include, for example, memory trace logs and rollback/restore logs.

(ii) **Execution.** Once ready, threads are created/assigned individual blocks of code and execution commences. During execution of the parallel section, each thread maintains a trace of the accesses to memory it performs. The number of accesses recorded can vary greatly dependent on the amount of analysis that can be performed statically, ranging from every access to only those of specific variables or even lines of code. Reads and writes are normally tracked separately so that they can be used to detect each different type of data dependence. Each thread often also maintains its own rollback or commit logs to be used in case a data dependence is detected, storing their own private copy of a memory location, or the original copy that they are modifying for use during the commit or rollback stages.

(iii) **Dependence Check.** After execution, or at suitable intervals during execution, the memory traces for each thread are analysed to detect data dependences.

(iv) **Commit.** After the dependence check has been performed and no data dependences have been discovered each thread's state has to be commited. Depending on how version control has been implemented this can involve writing back a thread's private copy of each memory location it modified, or it can simply involve discarding any rollback logs. After commit, each thread can be assigned more blocks to be executed speculatively, or if there is no more work to be executed then standard sequential or parallel execution continues.

(v) **Rollback.** Should a dependence be detected during the check then a rollback will be triggered allowing the program to return to a previous state of execution before the dependence occurred. Being able to rollback to a prior state is the primary means of ensuring that every speculative section executes correctly and produces correct results. Rollback methods also depend upon how rollback or commit logs are created, and can involve simply discarding a thread's private copy of a memory location, or something more complex such as replaying all memory accesses between threads in reverse to restore them to their original values.

(vi) **Re-execution.** After a rollback has been performed part or all of the speculative section is re-executed. Depending on the scheme this can be a safe, sequential execution, or an attempt at executing in parallel speculatively again. For instance, an iteration of a loop where a dependence was triggered by an earlier iteration that has now been committed may re-execute the current iteration again sequentially, or simply retry speculatively now that the earlier dependence triggering iteration has been committed.

(vii) **Teardown.** Finally once the speculative section has finished executing correctly with no data dependences speculative structures are often reset ready for the next speculative execution. Occasionally these structures can be destroyed however this tends to be inefficient in cases where more than one speculative section may exist.

As mentioned, most speculation schemes follow this same pattern; however, they can involve more complex operations allowing stages to be merged. For example, execution and dependence detection can be merged allowing for dependences to be detected *on-the-fly*. This can be beneficial if dependences are likely as it restricts the amount of wasted execution, however it also makes each thread's execution more complex and slower. Similarly dependence detection, commit and rollback stages can be merged however this requires specifically designed trace and version control structures to allow this.

The inter-thread ordering of each stage can also vary for each speculative model with the use of synchronisation to enforce ordering for each. There are two main types that exist, block-based execution, and pipelining. In block-based execution every thread executes the same stage or set of stages at the same time. If a thread reaches the end of its block before the other threads it must wait until they have before progressing. Alternatively, each thread could be pipelined with synchronisation preventing later threads to get ahead of earlier threads. For example, several threads could be executing, whilst another thread is performing its dependence check and another thread is commiting back to memory.

2.3.2 When to Check

The most important design aspect of a speculative scheme is the timing of the dependence check. This aspect influences every other aspect of the scheme, including the type and layout of the memory traces, the synchronisation methods used, version control types chosen and even how accurate the tracing needs to be.

This aspect is largely influenced by the type and requirements of the code being parallelised and the machine that the program is being executed on. For instance a loop that has a very low chance of containing a data dependence would likely benefit from a delayed and optimised or *lazy* dependence check, however using a delayed check requires that memory traces need to be stored for longer and are likely to consume more memory. In contrast, a loop with a higher chance of containing a dependence may benefit from *on-the-fly* or *eager* checking so that the dependence is discovered more quickly, however this requires more computation on each access restricting possible performance gains.

This section describes each method and analyses how they are likely to influence the other design aspects and performance of a speculative scheme.

2.3.2.1 Lazy Checking

For many loops and programs *lazy checking* is a suitable choice for when to perform a dependence check. In lazy checking each speculative memory access is logged and stored into memory until it is ready to be used to scan for data dependences. Then, at some later point during execution the traces generated are analysed to detect if a data dependence has occurred and, if so, handle the situation appropriately.

Isolating the memory-trace and dependence-detection stages tends to create a scheme that is simpler to implement. Each speculative access is often cheaper as it is less likely to interact with or share data with other threads thereby reducing the amount of expensive synchronisation required. With lower synchronisation costs the scheme will scale to higher thread counts more easily. Lazy schemes are also more versatile as the delay between tracing and detection can be fine-tuned to suit a given program and architecture. Additionally, the delayed dependence check can be optimised to perform as a batch scan of all logged accesses.

However, overall lazy detection is more taxing on memory. The size of the delay influences the size of memory access trace logs. Not only does a larger delay result in more accesses being recorded, but the structures used must also be more accurate in recording the address of each access to prevent *false* dependences from being detected. Longer time before a check also generates longer commit or rollback logs further increasing memory usage.

Besides higher memory usage, lazy detection also increases the cost of version control. For instance a longer rollback log increases the amount of memory that has to be restored if a dependence were to be found. Or it will increase the amount of time spent committing an execution back to non-speculative memory. Lazy detection is often unable to account for *safe* data dependences as the order in which accesses have occurred has not been recorded.

The added versatility and flexibility of lazy checking also adds complexity to the configuration of a scheme. Most notably is the selection of a suitable checking interval. Common intervals can include:

(i) **Iteration Based.** The dependence check is performed at the end of every iteration. This interval is used when a dependence is more likely to exist in a loop. It uses the least amount of memory for traces and version control and minimizes the amount of wasted computation in both execution and rollback should a dependence occur. It is, however, the most expensive interval in terms of synchronisation required between threads. It is also unsuitable to use during pipelined schemes as a way to hide critical sections in a scheme.

(ii) **Chunk Execution.** This interval, sometimes called a *sliding window*, performs the dependence check after a set quantity or *chunk* of iterations. This interval increases the amount of memory required, however is a good compromise between memory, synchronisation and version control costs. It is also suitable for pipelined schemes that use critical sections as the cost of a critial section can be hidden amongst the execution of each chunk.

(iii) **Extreme Laziness.** The dependence check can be left until the end of speculative execution, or until synchronisation points that are already inherently present in the parallel code. This method requires the least amount of additional synchronisation allowing for the fastest execution times. However, by leaving the dependence check until the last possible moment this method requires the largest mem-

ory traces, the longest dependence checks and the most expensive re-execution times. As an additional benefit, the complete isolation of the dependence check from speculative execution can allow for it to be performed independently of the rest of the program or even to be passed off to an auxiliary processing unit.

2.3.2.2 Eager Checking

For some loops or schemes it may be necessary to perform the dependence check on every memory access. This may be due to a higher risk of a dependence existing, or memory limitations such as being too small or slow to access. This is called *eager checking*.

On most architectures the use of eager checking is more expensive as it places the dependence check directly on the critical path of execution, increasing the amount of computation required for each memory access. Additional pressure is also added to memory by increasing the amount of information that is shared between each thread, limiting scalability. Additional synchronisation is required in the form of atomic operations or memory fences to ensure correct operation, further limiting scalability.

The memory usage of eager checking is often much smaller than that of lazy checking. Instant detection means fewer accesses must be stored in each trace, and shorter commit and rollback logs are generated. Similarly, the detection of a dependence is overall less costly than with lazy detection as less memory must be restored, and fewer iterations must be re-executed. Smaller traces may also be less accurate when storing the address accessed without detecting a false dependence. Eager checking can also correctly detect *safe* data dependences.

The main issue with eager checking is the limited scalability that it provides, but it has a smaller memory footprint, and with lower thread counts can provide a significant performance increase over sequential execution. However, due to the higher synchronisation and sharing costs, many schemes do not use a purely eager checking method.

2.3.2.3 Combined Checking

In some schemes the use of both eager and lazy checking is employed. This is normally performed as a compromise between computation required per access, the amount

of data shared between threads, the cost of rollbacks if a dependence occurs and the likelihood of *safe* data dependences.

2.3.3 Trace Topologies

The trace topology specifies the layout of each thread's memory traces and how they are stored and shared between threads. The topology influences the elemental type of the data structures used to store each trace and how the dependence checks are actually executed. The trace topology can also influence version control as rollback or commit logs can be stored in the same structure as memory traces. The specific topology for a scheme is heavily dependent on the timing of dependence checks.

Most, if not all, schemes record traces of reads and writes separately. This is due to the type of data dependences that can occur. If all accesses were stored in the same trace all dependences would still be detectable, however if two threads were to read from the same location then a false dependence would be detected, causing an unnecessary rollback. The positioning and type of each trace are not required to be the same allowing for more flexibility in the design of TLS schemes.

A possible alternative to using a read trace is using an access trace. The access trace logs all accesses, both reads and writes, allowing for the detection of RAW, WAR and WAW dependences in one comparison with another thread's write trace. This is a more efficient check, but requires the update of both the access and the write traces on every write performed.

This section describes the three main styles of topology of memory trace used in TLS, and analyse the conditions under which they are most frequently used.

2.3.3.1 Distributed Traces

Distributed, or single-writer traces, are ones that are assigned to individual threads and are only allowed to be modified by the thread that owns them. Each thread maintains its own trace with little or no communication with the other executing threads. During the dependence check distributed traces are exposed to the other threads to allow comparison with their own traces, detecting any dependences that may have occurred.

The single-writer model is one that minimizes the use of synchronisation whilst traces

are being generated. This allows faster recording of accesses during execution, hence an overall faster execution time. There is also no contention between each thread's traces allowing distributed traces to scale better than those that are shared.

Individual traces are generally smaller than shared ones as they are only required to store information about the thread that owns them; however, the use of distributed traces can significantly increase memory usage as every thread must maintain its own private copy. In general the memory usage of distributed traces tends to grow linearly with respect to the number of threads executing.

Due to the lack of sharing, distributed traces are normally associated with lazy dependence checking. During dependence checks it is likely that an individual thread's trace must be compared with that of all other threads causing quadratic growth in the overall number of traces to be compared. However, in certain schemes it is possible to distribute these traces amongst the threads or to use an alternative method of detection reducing this to only linear or logarithmic growth in number of traces to be compared per thread.

2.3.3.2 Centralised Traces

Centralised traces are those that are shared between some or all of the speculative threads. These traces can be read from or written to by any thread during execution and as such require explicit synchronisation to ensure each access is recorded correctly. Synchronisation can take many forms, but commonly it involves the use of locks, atomic operations or memory fences.

The overall memory footprint of centralised traces can be much smaller than that of distributed ones as there is only one trace that has to be stored. Increasing the number of threads does not increase the number of traces stored. The elemental unit of the trace may be larger in size to be able to distinguish between the accesses of each thread, whilst a distributed counterpart may need only a single bit to represent an access. However, centralised traces are most often used with eager checking, lowering the accuracy required before detecting false dependences, and lowering the length of the trace. This can counterbalance the larger element size by lowering the amount of elements stored, resulting in a lower overall trace size. Unfortunately as the number of threads grows, so must the accuracy of the trace, thereby increasing the number of elements stored, and resulting in a larger overall trace.

Centralised traces are often more complex due to the synchronisation methods they require. Their correct use also depends on the order in which they are read and updated and when the memory access is performed. Conveniently, when used with an eager checking method, the addition of extra speculative threads does not increase the amount of computation required for a dependence check. Instead, this complexity allows the check to be performed as part of the update to the trace. However, this benefit is rapidly negated at higher thread counts due to contention of the trace between each thread, severely restricting scalability.

2.3.3.3 Hybrid Topologies

Many schemes choose the use of a hybrid topology over purely distributed or centralised ones. In these schemes part of the memory accesses are recorded in a centralised trace, with other parts recorded using a distributed trace. For instance a scheme could use a centralised trace for recording each read access, and a distributed trace to record each write access. This method reduces contention on centralised traces allowing them to scale better, whilst maintaining a lower memory footprint than purely distributed structures. They can also allow for combined eager and lazy detection timings.

2.3.4 Synchronisation

To be able to correctly record and detect if a data dependence has occurred most schemes require explicit synchronisation at some stage during speculative execution. Most commonly this is done to ensure that each thread has finished generating its memory traces before it is used to scan for dependences, and also that each trace is not discarded before all other threads have performed their dependence check.

There are two main styles of synchronisation between threads: simply waiting for each thread to reach a joint stage of execution before performing a check, or pipelining each stage of execution and detection such that checks can be performed whilst other threads are still executing. Each of these methods can be implemented in a number of ways.

2.3.4.1 Barriers

The simplest form of synchronisation is to use a barrier to ensure all threads have reached a specific stage before they are allowed to perform a dependence check. This method does not allow for pipelining of each stage between threads and so is particularly useful when every thread has similar execution patterns. It is also particularly useful when the dependence check can be performed in parallel as sequential dependence checks would waste available processing resources and not be able to be hidden among other stages through pipelining. Too many barriers within code limit scalability, but when used sparingly these can produce significant speedup. The barriers may already be inherently present in the parallel code making them a suitable choice to trigger a dependence check.

2.3.4.2 Master Thread

A common way to implement pipelined synchronisation is the use of a *master thread*. During execution one thread, usually the lowest currently executing iteration, is assigned to be *master thread*. The master thread executes normally maintaining its memory trace. Once it finishes executing, its dependence check is performed. Assuming no data dependences are found, the master thread commits its changes back to main memory. Once completed the *master thread* status is passed to the next lowest iteration and the thread is assigned more work.

Meanwhile, all other threads are allowed to execute normally, maintaining their memory traces, but they are not allowed to progress on to their dependence check until they become the *master thread*. This method ensures that all prior threads to the master thread have finished executing, committed their results back to non-speculative memory and are guaranteed to be correct. Should the *master thread* discover a dependence, future threads are easily squashed, with any necessary rollback being performed followed by safe re-execution. However, this method introduces a critical section by only allowing one thread at a time to perform its dependence check and commit phases. This critical section can be hidden amongst other threads' execution stages, but this can severely limit scalability.

This may be partially alleviated by removing the critical section, and only using the *master thread* status as a barrier to ensure ordering between the threads. With a scheme

that allows for a parallel check and commit stage, ordering is enforced with two *master* statuses, the *check master* and *commit master*. Once a thread is ready to perform its check it must wait until it becomes *check master*. When it does, it immediately passes the *check master* status on to the next iteration. If a thread has not finished executing when it becomes *check master*, it waits until it has done so before passing the status on to the next iteration. Once the status has been passed the check is performed in parallel with any other threads at the same stage. The *commit master* status works in the same fashion, preventing a thread from committing until previous threads have finished their dependence check. This method effectively pipelines each stage of speculative execution, but can still suffer from scalability issues.

2.3.4.3 Overlaps

An alternative method is to keep track of which threads overlap without the use of locks or barriers. The most common way to do this is to record the wall-clock start and finish times of each iteration/block in a list and to perform the dependence check on any threads that have already finished executing that were executing or started executing during the current thread's life. This method is good for unordered commits as it does not require much in the way of locks or barriers, but for larger numbers of threads the maintenance of past and present memory traces can become unwieldy. Additionally, for ordered commits, such as loops, extra support is required to ensure they are committed in the correct order.

2.3.5 Trace Structures and Collision Detection

At the core of every TLS scheme are the data structures used to record memory traces. Each different data structure has different properties affecting, among other things, how easy they are to update, the amount of information each element is able to store and how they can be used to detect dependences. Which data structure is most suitable for a given scheme is dependent on whether they are to be used eagerly or lazily, whether they are going to be centralised or distributed, and what forms of synchronisation are going to be used to protect them.

This section lists some common data structures used to store memory access traces, analysing how they affect computation and memory usage. Some common ways to detect dependences using them are also discussed.

2.3.5.1 Bitsets

A common structure used to record a memory trace is a bitset or bitvector. Each bit in the vector can represent one specific memory address (or region) with 0 or *unset* repesenting an address that has not been accessed, and 1 or *set* representing an address that has been accessed.

Due to their simplicity bitsets are a dense data structure keeping their memory usage low. Despite their compactness, bitsets can still grow to an unweildy size if they are used to track larger memory regions without additional support.

Bitsets are relatively simple to update through the use of a shift and *bitwise-or* operation and can be used to highlight multiple locations in one operation simply. On most architectures each element is generally smaller than the elemental unit that can be accessed (e.g. char) therefore if they are used in a centralised topology explicit synchronisation would be required. However, due to the lack of information storable for each address they are not well suited for centralised use, instead being more suited as a distributed trace.

To most efficiently use bitsets to detect dependences is to directly compare them to other bitsets. To detect if two bitset traces contain conflicting accesses is to perform a *bitwise-and* operation on them. If any bits are set in the resulting bitset then a dependence has occurred. This method makes them well suited for batch scanning in a lazy fashion, as extracting individual bits is much less efficient. This makes them most suited for lazy detection schemes.

A less common use of bitsets is to use them to in a centralised structure per location to record which threads have read to or written from that location. Each bitset would be the length of the threads currently executing with each bit representing a single thread. This technique can be useful as it records a precise log of which threads have accessed which location providing an efficient method of finding the most recent or relevant copy of a location, for instance during value-forwarding techniques described in Section 2.3.7.4. This method is not suitable for lazy detection as the comparison of multiple locations requires the analysis of every individual location, rapidly becoming expensive on large regions. Additionally on large regions the number of of individual bitsets can become unweildy.

2.3.5.2 Counters

Larger elements are a frequently used structure in memory traces. Instead of using a single bit to represent each memory location, larger elements instead use, for example, an array of integers or chars to represent memory locations. This allows for a greater flexibility in how they are used, however this generally also adds additional computation and memory requirements.

Having larger elements allows the use of booleans as an alternative to bitsets, removing the need of shift and other bitwise operations to update each location. Instead they can be directly addressed in the array, allowing for use as a centralised trace without the need for explicit synchronisation on every access. However, as with bitsets, the lack of information provided by booleans make them generally unsuitable for centralised traces.

Instead, the use of counters is more common. Frequently each element is used to store the highest iteration ID to access the location or region represented by the individual counter. On subsequent accesses, or during commit, the value stored is used to determine if a later iteration has already accessed that location. For a read log a dependence violation occurres if a higher iteration has already written to that location, and for a write log a dependence violation occurs if a higher iteration has already accessed the location at all. This method is only useful when used as a centralised trace, however can be used with both eager and lazy checking schemes. The eager use of this method can also be used to distinguish between safe and unsafe data dependences.

Another use is to count how many threads have accessed a location in a centralised trace. If multiple iterations have read from a location, and at least one thread has written to it then a dependence has occurred. Similarly if multiple threads have written to a location then a dependence has occurred. This use is conceptually simple, however it requires extra support, possibly in the form of distributed traces, to allow for the same thread performing multiple reads or writes of the same location, or some combination of the two.

Using each element as a counter increases the complexity of accessing them, as they are frequently required to use minimum or maximum operations, or other arithmetic operations to update them. A centralised counter must also use explicit synchronisation such as atomic operations or memory fences to ensure their correct operation. Finally they are also not well suited for updating a range of elements, nor are they optimised

for bulk scanning as they all must be accessed and updated individually.

Counters are occasionally also used for version control to identify and order backup copies of memory locations, incrementing their counter every time a location is preserved.

2.3.5.3 Address Lists

One of the simpler structures used to track memory accesses is to create a list of all the addresses that are accessed. To update such structures you simply add the address accessed to the back of the list making it very lightweight in terms of computation per access. However due to the sheer quantity of accesses performed in most speculative code they are likely to grow in size quickly and become unwieldy. They are also more commonly sorted in order of access making them difficult to compare to each other directly.

Alternatively, when used in combination with other data structures, address lists can become more useful. For example, a centralised counter-based read log can be used to compare against a write-address list. Each element in the list is examined for the current iteration and compared to the read log. If a higher iteration has read a location in the list then a dependence has occurred. Using the list can decrease the amount of computation and synchronisation required per write during execution. However, traversing the entire list can be expensive if the number of writes is large and writes to the same address will cause wasted computation.

2.3.6 Accuracy and False Dependences

Another factor affecting speculation is the granularity of memory traces. In efforts to reduce memory usage and increase speed various techniques such as reducing the access addressing size or by hashing individual accesses have been developed. Using these methods can affect the accuracy of the dependence detection often triggering false dependences thereby reducing performance. In an effort to restrict this effect the use of per-variable structures can also sometimes be used. This sections discusses each method for reducing memory usage and ways to alleviate the effects of doing so.

2.3.6.1 Addressing Size

To lower memory usage required for memory traces the base addressing size for tracking can be modified. In some circumstances every bit modifiable would be able to be tracked however this is frequently not required. Ideally the base unit tracked would be the same as the smallest addressable unit of the CPU or memory system being used, allowing for every write performed to be scanned for dependences. However, doing so results in large data structures being required and selecting a larger unit can shrink this. For example tracking at a word level would result in a trace structure that can be either four or eight times smaller, dependent on the machine, over that of byte level tracking. However if each thread performs accesses on a byte level instead of a word level, tracking only the word address could easily cause the detection a *false* dependence and trigger an unnecessary rollback. Going further than that, tracking at the page level would result in a tracking structure of over 4000 times smaller, assuming 4 KiB pages, however this would almost certainly result in the detection of false dependences.

The guaranteed minimum size of tracking can be determined easily at compile time based on the minimum size of all speculative accesses. However, the use of static analysis may be able to improve on that by proving that a thread will perform a series of access to subsequent memory addresses.

2.3.6.2 Hashing Techniques

An additional method of reducing the size of trace structures is to store only the hash of the address being accessed. For example, a program using 32-bit byte-level addressing on a 1 KiB array does not need to store the entire 32-bit address. Doing so would require a trace structure capable of storing/distinguising between over 4 billion different entries. Instead, to detect dependences for just the array only 1024 entries are required. A simple hashing scheme for this would be to record only the least significant 10 bits of the address and discard the rest.

The example given above is a rather simplistic one. Frequently the sizes of the memory ranges being tracked are unknown at compile time, are much greater than 1 KiB arrays and are required to cover multiple different arrays during execution. Hashing can be a very useful tool to reduce the size of the trace structures however the ideal sizes of these can be difficult to determine. Hashing to too many items creates a large, wasteful and inefficient structure that can harm performance. Hashing to too few creates a

small and efficient trace, however significantly increasing the chance of a detecting false dependence.

2.3.6.3 Single or Multiple Traces

As mentioned in the previous section, most speculative tracking is performed on more than one memory range, be it a individual variables, multiple arrays, pointers or simply some random block of memory that the compiler cannot determine what it is being used for. There are multiple ways of handling these situations, namely the use of single or multiple traces.

Single traces record all speculative accesses into the same trace. Note this does not mean a centralised trace as mentioned in Section 2.3.3.2, but instead a single structure covering the entire address space of the program. Conversely multiple traces record speculative accesses into a separate trace for each memory range they are accessing. For multiple traces, in a centralised scheme there would be multiple centralised traces, one for each range, and for distributed traces each thread would have multiple traces of its own for each memory range.

The use of single traces is generally the most simple to implement and most convenient for detecting dependences. When using single traces every speculative access is automatically tracked correctly and any dependences that can occur will be detected. For instance if pointer aliasing is possible in the program a single trace would still detect dependences when these variables were accessed. Similarly single traces would have no issue if arrays were to overlap or some other similar issue. However the use of single traces complicates the required hashing techniques. For example instead of dealing with a single 1 KiB array, two 1 KiB arrays could be being tracked. If the arrays were stored in adjacent memory this would not be an issue, simply set the hash function to allow for 2048 addresses. However if they are stored in non-adjacent locations, potentially far away from each other then the hash function must account for this to prevent the detection of false dependences.

To ease the issue of hashing, instead multiple traces can be used with independent hash functions. Each function is only required to distinguish between 1024 entries no matter how far apart the two arrays are. However, this method is more complex as it requires all tracking methods to know which trace to use. Additionally aliased pointers must also know which trace to use, which might not be the same on each

access. If the incorrect trace is used then existing data dependences may go unnoticed defeating the purpose of speculative execution. Similar issues occur for overlapping arrays. Finally, the use of multiple traces also means that all traces must be scanned for data dependences increasing the number of checks required.

2.3.7 Version Management

Version management is the underlying protection provided by TLS that allows for correct execution. During execution and most importantly after dependences have been detected it is version management that allows the program to roll back to an earlier state and re-execute in a safe manner.

Often the version management is relatively separate from the memory traces and dependence detection. Backup or restore mechanisms often use their own structures for keeping track of what has been backed up and store commit or rollback state in their own private buffers. However, occasionally memory traces and version management are combined such that commit or rollback values are stored inside memory trace structures. A prime example of this would be using an address list and to store the rollback or commit value in the same list.

Whether they are combined or not, the timing of the dependence check is intrinsically linked to how much and for how long rollback or commit information is stored for. For example a purely eager scheme would be able to discard of rollback information or perform a commit as soon as all previous iterations have completed, whereas an extremely lazy scheme would have to maintain a rollback or commit log for the entire speculative execution.

The type and granularity of version management used has a large impact on the performance of a speculative loop. This section lists several methods, and analyses how they impact the overall performance of speculative execution.

2.3.7.1 Management Scheme

The type of version control used to protect speculative memory regions varies greatly in terms of how it is implemented, with additional options for when to perform version control, where it is to be stored and how to store it. Each different option affects

the performance, memory footprint, checkpointing ability and scalability of speculative execution. There are two main categories of implementing version control for speculative execution: rollback or commit.

Rollback. Speculative memory regions are backed up as necessary to another part of main memory. When a dependence is detected the backup copies are then restored to their original location to allow for safe re-execution. If no dependence is found then the backups are simply discarded as they are no longer needed.

Commit. Any modifications to speculative memory regions are stored in a private buffer during execution. After the dependence check, if no dependences have been detected then the privatised buffer is commited over the original speculative memory regions ready to continue with the rest of the program. If a dependence is found then the private buffers are discarded and re-executed.

It is intrinsically clear that the rollback management scheme favours execution that is unlikely to contain data dependences as there is no additional commit phase required. The commit management scheme instead favours execution that is more likely to contain a dependence as re-execution does not require an expensive rollback stage.

2.3.7.2 Versioning Schedule

After the overall scheme for version control is decided a versioning schedule must be chosen. The versioning schedule affects how large version-management data structures become and how they are to be interacted with. Again, two main choices exist for versioning schedules: preemptive and just-in-time.

Preemptive. At the start of a speculative stage backups or private copies of all speculative regions are created. This can be costly to do so as each region may be large and the speculative execution may only touch a small part of each, however this is a one-time cost at the start of a speculative section. There is no additional version control required during the execution stage of speculation. This method is also restricted in that it only maintains the single checkpoint at the start of the speculative section significantly increasing rollback and re-execution times if a dependence is detected. If used with a commit based scheme, preemptive scheduling may require additional support to track which locations in the speculative region must be committed.

Just-in-time. On every access, particularly on writes, the location accessed is backed up or privatised. This has the benefit that version control is only used on locations that are actually touched, however involves additional computation on every speculative access. Extra support must also be integrated to keep track of which locations have already been backed up or privatised, which must also be checked on every speculative access. However, JIT scheduling has finer grained control on when checkpointing occurs, allowing for shorter rollbacks and re-execution.

2.3.7.3 Data Structures and Topology

Besides the scheme and schedule, version control also has a variety of options for what kind of data structure to use to store rollback or commit logs. Through its nature schemes using preemptive version control only has the option of copying the entire speculative region to another section of memory, thereby creating a shadow array that is filled from the start. In constrast JIT scheduling schemes are much more flexible, and generally have the option of using a shadow array, a hash map or a list to store rollback or commit logs.

Shadow Array. By using a shadow array, backups and privatisation can be performed in a low constant time, making them a reliable option for performance. However they have the largest memory footprint of all models as they have to account for all possible locations being accessed.

Hash Map. To reduce the memory footprint of version control a hash map can be used. The access times of a hash map are similarly constant however involve more computation on every access. They also have the risk of collisions occurring in the hash function which must be handled correctly.

List. The fastest structure for insertion is to use a list, implemented as an array, a linked list or some other form. Each new item is simply added to the end of the list, stored as an $\{$`address, value`$\}$ pair. Extracting values on subsequent reads or writes are more complex as the list must be traversed to find the relevant item. This can be fast if it was accessed recently, or slow if not. The size of lists can also be unpredictable as they grow based on the number of accesses.

Process Isolation. To mitigate the overheads associated with accessing speculative

structures some schemes use process isolation as the primary means of version control. By forking the main thread version control is delegated to standard OS operation, implementing copy-on-write semantics for any page modified. This removes any costs associated with the lookup of values already in speculative structures, beyond the cost of the OS creating a new copy of the page. This is normally used in conjunction with write lists or some other structure to allow simple transfer of speculative state back to the main process.

The topology of each data structure is similarly important, but is generally based upon the scheme and schedule of version control. Specifically, backups or privatised versions can be stored centrally or distributed between each thread.

Central data structures make the most sense for preemptive schemes as only one checkpoint needs to exist. Distributed data structures are more appropriate for schemes when multiple checkpoints exist simulataneously as each thread can maintain its own state.

2.3.7.4 Value Forwarding

For some schemes it is possible to employ the use of value forwarding to mitigate the cost of some types of data dependences. When each thread maintains its own private copy of a variable that it has written to, a later thread or iteration can directly access that private copy for their own use converting some RAW dependences into *safe* data dependences. The ordering during access is important to ensure that further writes by the earlier trigger a dependence violation correctly, hence this technique can only be used by eager checking schemes. Additionally the support code required to perform such forwarding adds additional computation complexity and requires customised data structures to enable the discovery of the correct version of a variable.

2.3.7.5 Granularity

Finally, similar to the granularity of memory traces, version control can be handled on different data sizes. This is most relevant for JIT schemes as preemptive schemes work on the entire speculative region. The most common choices are word-based and page-based version control.

Word-based version control allows for fine accuracy between each thread, allowing for very controlled rollbacks. However they require the most amount of computation

for each access and backups or privatisation is common. Alternatively, page-based version control involve less frequent backups and can be optimised better due to the larger amount of data stored each time. Using page-based version control lowers the accuracy thereby restricting checkpointing.

2.4 Conclusions

This chapter has provided an overview of the common techniques required for parallel execution. Listed was a summary of standard parallel execution schemes, followed by an analysis into the ways to automatically implement some of those schemes. Finally an in-depth analysis of common speculative execution techniques was provided with analysis into their performance and resource-utilisation influences.

As these are only descriptions of commonly used methods the next chapter will provide a look at some existing tools that use the techniques described here.

Chapter 3

Related Work

Due to the poor performance of static analysis alone [7], modern research has focused on using profiling techniques to discover parallelism within programs [44]. As this technique is inherently unsafe, many processes have relied upon user verification as a final guarantee that a section of code is safe to run in parallel. This verification completely rules out use in a fully automated parallelisation system; therefore much research [29, 28, 24, 25, 36, 30] has already been performed into speculative execution schemes which can facilitate fully automated parallelisation.

This chapter presents an overview of previous research into the field, first with an analysis of an existing profile-based parallelism detection in Section 3.1 and then a selection of existing speculation schemes. First, purely CPU-based schemes are presented in Section 3.2, followed by a purely GPU-based speculation scheme in Section 3.3. Final conclusions are presented in Section 3.5.

3.1 Profile-Driven Parallelism Detection

In 2009 **Tournavitis and O'Boyle** [44] presented an aggressive method for discovering and exploiting parallelism in sequential programs. Their method uses a two-stage approach that uses profiling techniques, to discover unexploited parallelism and then a machine learning (ML) technique to identify which of these targets would provide increased performance when executed in parallel.

The first stage of parallelisation involves instrumenting sequential programs at the in-

termediate representation (IR) level of the CoSy compiler framework [1]. This instrumented code is then executed using one or more sample input data sets, with the memory access traces of the program recorded to a separate log. Performing this intrumentation at the IR level allowed them to maintain more structure about the program being executed than would otherwise be possible using standard profiling tools, such as start and finish of loops and their individual iterations or the start and finish points of calls to functions and methods.

Once a memory trace has been collected their approach then performs a series of static and dynamic analyses on the code and logs to identify definitely parallel, probably parallel and definitely sequential sections of code, focussing mainly on loop parallelisation. The analysis performed also identifies variables that require privatisation, reduction operations that may occur and critical sections and synchronisation points that may be required during parallel execution. Privatisation and reduction identification is able to be performed beyond individual variables up to complete arrays. Parallel code for each loop is then generated automatically for each of the stages.

Stage two of their approach uses ML techniques to identify profitable loops of those that have been targetted. The ML technique uses an offline supervised learning scheme whereby a number of loops are executed both sequentially and in parallel with their execution times recorded. Next, loop features are extracted both statically from the code and dynamically from the profiled execution log which are combined with some features regarding the hardware used to execute the loop. This data is then used to train the ML-based predictor. Targetted loops have the same features extracted from them which are then used by the predictor to determine if a loop is likely to be profitable. Likely profitable loops are then presented to the user for the final decision over if a loop is safe and profitable to execute in parallel and those selected are incorporated into the parallel version of the program.

This technique presents highly likely and profitable candidates for parallel execution however, due to the inherent nature of the profiling stage, it is unsafe for use in an auto-paralleliser. The only suitable way to use this technique in completely automatic parallelisation scenarios is to employ the use of speculative execution over each probably parallel loop. Additionally the ML-based profitability predictor must also be extended to account for the costs of speculative execution.

3.2 CPU Schemes

3.2.1 S-TLS

In 2001 **Rundberg and Strenström** [36] presented one of the first fully capable purely software based speculative schemes, S-TLS. This scheme was based very closely to existing hardware techniques by including a speculative data structure for every memory location covered by speculative support. Each memory location came with an associated load vector, store vector, a local *shadow* copy of the variable per thread and a lock used for every access of the location.

The load and store vectors for each location were implemented as a bitset the length of the number of the executing threads. On each load the load vector bit belonging to that thread was set, and then the entire store vector was compared to determine if a lower thread had already written to that location. If so, the most recent local shadow copy of that location belonging to a thread lower than the current was used allowing execution to continue, otherwise the value in non-speculative memory location was used. On a store, the appropriate store vector bit was set, then a local copy of the written value was stored in the shadow variable for that {`location, thread ID`} pair. A check was then performed to identify if a higher thread had already read from the location and if the higher thread had obtained its value from a shadow variable higher than that of the current thread. If not, a rollback was triggered by clearing the shadow copies and resetting the vectors, followed by sequential execution of the speculative section. At the end of speculative execution, the shadow copy belonging to the highest thread of each location written to was commited back to non-speculative memory.

This scheme provided an eager check allowing unsafe dependences to be detected when they occurred. However, every access was performed using an expensive locking scheme, completely blocking all other threads from accessing variables simultaneously. The use of value forwarding described above also mitigated the possibility of WAR and WAW dependences from causing speculation to fail. The use of a structure for every location tracked also created completely accurate detection, disallowing the possibility of false dependences. However, all of these features combined caused an extremely high memory footprint associated with the model, and the commit process required every speculative location to be checked to determine if it required committing. While this could be performed in parallel, it could be extremely time consuming

if there was a large memory region covered by speculative protection.

3.2.2 PolylibTls

Between 2007 and 2009 **Oancea and Mycroft** released several speculative execution schemes, along with a related library, to aid implementation of speculative execution in loops. The library was geared towards a new method for programmers to write possibly parallel loops, rather than automatically detected parallelism. However, with static analysis and various code transformations this feature could be introduced relatively easily. Of particular interest were the speculative schemes proposed and evaluated along with the library: SpLSC, a commit-based model described in Section 3.2.2.1, and SpLIP, a rollback-based model described in Section 3.2.2.2.

3.2.2.1 SpLSC

SpLSC [28] is a *Serial Commit* speculation scheme using a lazy detection checking schedule. Their scheme introduced the use of hashing to reduce the memory overheads associated with memory tracking structures, and the use of a transferrable *master thread* to aid synchronisation.

In SpLSC each thread maintains its own commit log, implemented as a series of {address, value} pairs stored as a simple array. During execution this log is filled and then committed back to non-speculative memory one thread at a time. To alleviate the sequential nature of this commit sequence, it is performed in a round-robin pipelined fashion every n iterations by the master thread, whilst the other threads continue to execute the loop, filling their own logs. During optimal configuration and performance this method effectively hides the commit cost amongst the execution of all other iterations (with the exception of a setup and destroy cost per loop). However misconfiguration or inconsistent iteration execution times can cause this commit time to become apparent and to become a detriment to execution times. Its serial nature also limits this scheme's scalability.

As a tracking structure, SpLSC uses a single, centralised vector to record which threads have read a given memory location. Elements within the vector are counters that contain the highest thread to read a given variable at any given moment, updated using atomic instructions or lock-free based accesses. Each element corresponds to a

hashed memory address, allowing multiple addresses to use the same element. This drastically reduces the required size of the vector, but opens up the possibility of false dependences should the user-specified hash function have unfortunate collisions. Additionally, each thread maintains a local bitset to record if it has written to a given location, used during reads to determine whether the read must come from the commit log or directly from memory, avoiding thread-local RAW dependences.

During commit, all writes in the commit log are written back to non-speculative memory in the order which they were performed, avoiding all WAW dependences. On each commit, the address is re-hashed and the load vector for that location is examined. If a higher thread than the master has already read from the location being committed to, then a dependence has occurred, all higher threads' commit logs are discarded and re-execution commences. This method ensures that only work performed by threads after they pass on the *master* status must be re-executed.

This scheme has a low memory footprint when configured correctly and can provide a sizeable speedup. However if it is misconfigured there is a high risk of false dependences. There is also the risk of slowdowns compared to sequential speed if the commit phase is not hidden by iteration execution times. Similarly, the scalability of this scheme is severely limited by the bottleneck introduced by the sequential commit stage and also by the use of the centralised load vector.

3.2.2.2 SPLIP

SPLIP [30] is an *in-place* speculation scheme using an eager dependence detection schedule. This scheme was developed with the aim of prioritising fast execution over quicker rollbacks, and to alleviate some of the scalability issues presented by SPLSC. This scheme also uses the *master thread* for synchronisation, and hashing techniques on the speculative structures to minimise their memory footprint.

In SPLIP all writes to speculative memory are performed *in-place*, with each thread maintaining a rollback log for the writes that it has performed as a series of {`address`, `value`, `version`} triples stored in a simple array. As with SPLSC each thread executes *n* iterations, filling the rollback log before waiting to become *master*. Once a thread receives this status and has not detected any dependences, then all of its writes have been proven to be dependence free and as such the thread can discard its rollback log, proceeding to execute the next *n* iterations. If a dependence is detected by a thread,

it waits to become master, then triggers a rollback of writes of its own and subsequent threads.

To implement the in-place writes and rollback features, the speculative structures used are more complex than those of SPLSC. In addition to an identical load vector, there is a functionally equivalent write vector, a stamp vector used to versioning of stored elements and a sync vectors used to alleviate the need for atomic instructions. On a read the load vector is set to the maximum of the current value or the accessing thread, the read is performed and then the store vector for that location is compared to identify if any higher thread has written to the location. If so a dependence has occurred. On a write the store vector is set to the maximum of itself or the current thread ID and examined to determine if a higher thread has already written. Again, if so a dependence has occurred. The current value is backed up to the rollback log along with a version ID for the write taken from the stamp vector, which is also incremented. Finally the write is performed and the load vector for that location is examined. If a higher thread has read from that location then a dependence has occurred.

Rollbacks in SPLIP are complex and expensive. As each write can be performed in any order the original value of the location can be stored in any thread's rollback log. Additionally, as the versioning system relies upon a hashed location every write must be replayed in reverse to ensure that all locations are restored to their original value. This process must be performed sequentially whilst no other threads are executing.

This scheme presents an effective way of minimising the sequential phase that limited scalability in SPLSC, allowing the scheme to scale to higher thread counts before performance degrades. Unfortunately, the use of the transferrable *master* status still introduces a serial bottleneck between threads, limiting scalability. The structures used to store speculative state are much larger than that of SPLSC and are still subject to the possibility of false data dependences when misconfigured. Additionally the large use of centralised data structures introduces a lot of pressure on the memory system with large quantities of data being shared constantly between each thread, limiting scalability further.

3.2.3 STMLITE

In 2009 **Mehrara and Mahlke** released STMLITE [25], mainly for use in transactional memory systems such as databases. However, despite this their work included and evaluated an extension to support loop writeback ordering to allow its use for loop-based parallelism. STMLITE is a lazy, hybrid decentralised detection scheme.

In contrast to previous schemes, all dependence detection is performed in a separate thread and checks are performed based on overlapping execution of individual workloads. For loop-based parallelism, a workload corresponds to a block of n iterations. Speculative storage for each workload is stored within a transaction log that stores the start and finish times of the execution, along with read and write signatures, a page-based commit log and other minor state values. At the end of execution, the transaction log is passed back to the commit management thread, which maintains a list of previous and uncommitted transactions. The read and write signatures of the submitted log are compared to the write signatures of any uncommitted logs and any previous logs that overlap in execution. If there is a collision, the log including any waiting commits are discarded and re-executed, otherwise the thread is allowed to write back any commits that are waiting and the log is added to the previous list. Loop support is added by ensuring checks cannot be performed until the workloads of previous iterations have finished their checks.

Read and write signatures are implemented as hashed bitsets. Every address is hashed to a particular bit within the bitset, with a set bit indicating that an address hashing to that location has been accessed. By having each thread maintain its own private copy of a read and write signature, each thread is allowed to execute with minimal synchronisation overheads, and lower memory contention as compared to previous schemes. The hashing of memory addresses also lowers the memory footprint of each transaction log significantly, but again opens up the possibility of false dependences being detected. The use of a separate thread to perform checking also introduces a bottleneck between the threads that limits scalability.

3.2.4 DSWP & SMTX

In 2008 **Raman and August** proposed Decoupled Software Pipelining [35] (DSWP) as a more complex method for extracting parallelism from sequential programs. This

method was proposed to allow for earlier resolution of known data dependences to allow additional threads to start executing their bulk work sooner. This was performed by splitting a loop iteration down to basic blocks separated by cross-iteration dependences. The initial thread would execute the first block until all dependences for the next iteration have been resolved. The results of the first block are then passed onto the next thread which executes the remainder of the iteration whilst the initial thread would then repeat the process for the following iterations until there is no more work to be done. Meanwhile, with the cross-iteration dependences already resolved, the bulk of the iteration processing can be performed in parallel. This method is particularly useful when executing DOWHILE loops, for instance traversing a linked list, by allowing the loop conditions to be calculated early on in cases where parallel threads would normally be delayed from executing. As an added benefit it allows for latencies associated with forwarding the dependent values between threads to be hidden by the pipelined execution of basic blocks.

DSWP can suffer from similar limitations to those associated with profiled parallelisation, particularly due to the limitations of static analysis where a dependence does not exist but cannot be proven so. In cases of these *potential* dependences similar speculation techniques can be employed to allow for parallelisation, hence in 2010 an extension to DSWP was released to allow this, Software Multi-threaded Transactions [34] (SMTX). Notable to this form of speculation is its use for detecting dependence violations solely cross-iteration, not cross-thread as each iteration is processed across multiple threads.

SMTX is a commit-based, lazy-detection speculative scheme using a single thread to process conflict detection and final commit of speculative state. Each transaction, executed across multiple threads, generates a separate write log which is passed between each thread processing that transaction and finally to the commit thread. Each executing thread is separated using *process isolation* to mitigate the costs associated with accessing speculative memory, and must therefore replay these write logs at the start of each basic block to update their private view of memory. Rollback is performed by restarting processes that caused a dependence, therefore forcing them to re-obtain their view of memory from the non-speculative thread.

SMTX does not specify a particular conflict-detection scheme, delegating that choice to the programmer/compiler. However, the use of a single thread for conflict detection and commits ultimately limits the scalability of this scheme, and the transfer and

replaying of write logs can add significant overheads for long-running or many-stage transactions.

3.3 GPGPU Speculation

General purpose GPUs (GPGPU) are special purpose processors with many cores, in the range of hundreds, and likely to expand into the thousands as time progresses. They have much more stringent restrictions on processing and access to memory that make them much more difficult to program for even by an experienced parallel developer, let alone a standard auto-parallelisation tool. With the advent of more and more general purpose GPUs becoming available in all devices, ranging from phones and tablets, to standard computers and laptops, to supercomputers and clusters it is desirable to be able to exploit this massive processing power. Due to the difficulties associated with programming for them it would be beneficial for this process to be automatable. As with CPUs, limitations exist with static analysis for GPUs restricting the ability to extract parallelism for these devices automatically. Therefore, more aggressive parallisation techniques must be used and hence, speculative execution on a GPU.

Most research on speculative execution has focused specifically on executing parallel programs on the CPU. To date only one viable scheme has been proposed that effectively utilises speculative execution on a GPU.

3.3.1 PARAGON

In 2012 **Samadi and Mahlke** released a collaborative CPU- and GPU-based parallelisation scheme, named PARAGON [37], that utilises both the CPU and GPU simultaneously. The inherent limitations of GPU execution makes on-device rollback and re-execution illogical, as the cost of doing so just once would likely make the entire program execute slower than doing so sequentially. As an alternative, PARAGON splits a program into three types of blocks: sequential, parallel and speculative. Sequential blocks are executed solely on the CPU, suitable parallel blocks on the GPU and suitable speculative blocks on both. If the GPU version executes faster than the sequential CPU version and does not encounter any data dependences during execution then the CPU thread is stopped and the results from the GPU are used. Should the GPU version

take longer or encounter a dependence then the CPU version is allowed to finish and its results are used for the rest of the program.

Version control does not require additional processing on the GPU, as it is inherently handled by maintaining two versions of memory during all speculative execution: one on the CPU and one on the GPU. The CPU version is considered non-speculative, whereas the GPU version is speculative. On a successful GPU execution, speculative memory is copied back to non-speculative memory, overwriting any work performed by the CPU. On a failed execution, speculative memory is recovered from non-speculative memory at the start of any GPU based execution.

Conflict detection is performed by the GPU code by maintaining two vectors, a read vector and a write vector. Each element of these vectors represents an address accessed and stores a count of how many writes were performed to that address, updated efficiently using an atomic increment function, or a simple boolean of whether a read has occurred. Address hashing is not used by PARAGON. At the end of speculative execution the entirety of these vectors are processed using a separate kernel. During processing, if an element has multiple writes or has been read and written to then a dependence is triggered.

The run scheduling of this parallelisation scheme is efficient and well designed, ensuring at worst a sequential runtime of the program plus minor overheads associated with GPU management. However, the conflict detection scheme used is naive, and has a very high probability of triggering false dependences, since iteration-private dependences are detected as errors but only cross-iteration dependences would cause an actual error in the final output. In many speculative execution scenarios a thread reads a value from an array, performs an operation on the value and then restores the value back to the original array (iteration-private WAR dependence). With PARAGON this would be detected as an unsafe dependence, when no such error occurred. Similar false dependences would be detected for iteration-private RAW and WAW dependences. The use of a delayed vector-processing kernel also causes the execution times to become worse as the entirety of each vector must be processed, not just the elements accessed. For small vectors this is likely not an issue, however larger vectors would have a significant impact on processing. Should address hashing be used in an attempt to shrink the size of these vectors the risk of false dependences would only become worse.

3.4 Hardware Based Speculation

Purely hardware based speculation schemes can reliably provide significant speedups to a given program. With some designs this speedup can also be provided automatically in hardware increasing further increasing its appeal. However due to its nature HW-TLS requires customised hardware to support exectution, increasing the difficulty and cost of development. As such there are no feature complete versions of HW-TLS commercially available. The nearest implementation is Intel's Haswell [15] architecture through the use of Transactional Synchronisation Extensions (TSX).

3.4.1 Transactional Synchronisation Extensions

As can be inferred from the name, TSX applies a more Transactional Memory based approach by extending existing cache coherence protocols to record whether or not a cache line has been accessed during a transaction. Naturally this provides a detection granularity size of a single cache line. TSX also provides two new programming interfaces:

Hardware Lock Elision (HLE)

HLE exploits the idea that mutex lock and unlock operations are implicit transaction boundaries and that the granularity of locking is often larger than that of available parallelism due to tradeoffs of code complexity and locking costs. Instead of locking a region of code HLE records the address of the lock and attempts to perform all instructions within the locked region as a transaction. If the transaction fails then the lock is actually aquired and the code is rexecuted.

Restricted Transactional Memory (RTM)

RTM provides additional processor instructions to denote the start and end of transactional regions. Traditional transactional memory simply discards failed transactions and reattempts them, however this can introduce livelock, priority inversion and starvation issues. Intead RTM takes an address at the start of each transaction for the program to jump to should a transaction fail, delegating the consequences of transaction failure to the programmer.

Haswell and TSX have some inherent limitations. As with all hardware schemes there are physical limits on the quantity of speculative state that can be stored, beyond which

the transaction will fail. In the case of Haswell's implementation this is the 32KiB L1 Cache. The granularity of speculation is also fixed at the length of a cache line, 64 bytes. It is also transactional in nature restricting the automated parallelisation of loops without additional software support. Such support would include restricting the committal of later iterations until transactions containing earlier iterations have succeeded.

3.5 Conclusion

This chapter has reviewed an aggressive method for auto-parallelisation and various existing schemes for implementing software-based thread level speculation on both the CPU and GPU, as well as a hardware based transactional memory system.

Tournavitis' aggressive auto-parallelisation technique is a reliable method to extract parallelism from sequential programs however its use relies upon user interation to add safety, hence the need for TLS. Their machine-learning-based profitability and mapping scheme is useful for determining when to parallelise a loop, however its use is fine tuned towards standard parallelism and not to SW-TLS therefore it would be desirable to have a similar scheme extended to select and configure a speculation scheme for each parallel region when it is suitable to do so, taking into account the many costs incurred by executing a program speculatively.

Many existing speculation schemes focus heavily on the ability to rollback but in many circumstances doing so causes a program to perform worse than the original sequential version. When using the profiling techniques described it is possible to be virtually certain that a loop is safe to execute in parallel. With the knowledge that a single rollback would ultimately defeat the purpose of parallel execution and with such reliable parallel candidates available it is desirable for a scheme to exist that does not focus so heavily on the ability to rollback and can instead optimise purely for parallel execution performance, with the potential for an extremely expensive but highly unlikely rollback cost.

Additionally, GPU speculation schemes are currently in their infancy and whilst increased performance has been proven to be possible the current detection scheme are naive and inaccurate. It is highly desirable for new GPU schemes to be developed that incorporate some of the methods and features of CPU schemes to add more reliability

and flexibility to GPU based software speculation.

Finally the hardware scheme is limited in nature with respect to speculative state and requires additional support for loop based software speculation.

With a clear understanding of the techniques commonly used in SW-TLS, the next chapter presents a new SW-TLS scheme that attempts to fill the gap where other schemes perform poorly.

Chapter 4

Lightweight Pipelined Speculation

During testing of several speculation schemes implemented using PolyLibTLS [29] several performance limitations and bottlenecks of the schemes became apparent. This chapter proposes and evaluates a new lightweight pipelined speculation scheme (SpLPC) to supplement the existing schemes SpLSC and SpLIP.

Upon investigation the *critical* checking and commit phase of SpLSC performed well when its costs could be hidden amongst the execution of the other threads; however, this phase frequently takes too long to execute for its costs to be hidden. In these circumstances other threads finish executing their iterations and are forced to wait for the previous threads to finish committing. At longer commit times this problem cascades such that every thread spends much of its time waiting to commit, and none of the program is truly executed in parallel.

SpLIP is an in-place model that relies heavily on the use of centralised load, store and versioning vectors to implement its dependence check and rollback functionality. Unfortunately, as the number of threads executing rises so does the number of threads accessing each of these vectors simultaneously. As this occurs each vector becomes thrashed, causing each speculative access to take longer time, limiting the scheme's ability to scale to higher thread counts.

This chapter proposes a new SW-TLS scheme, Lightweight Pipelined Commit (SpLPC) with the aim to:

(i) remove bottlenecks caused by sequential checking and committal,

(ii) alleviate cache thrashing by using thread local load/store vectors.

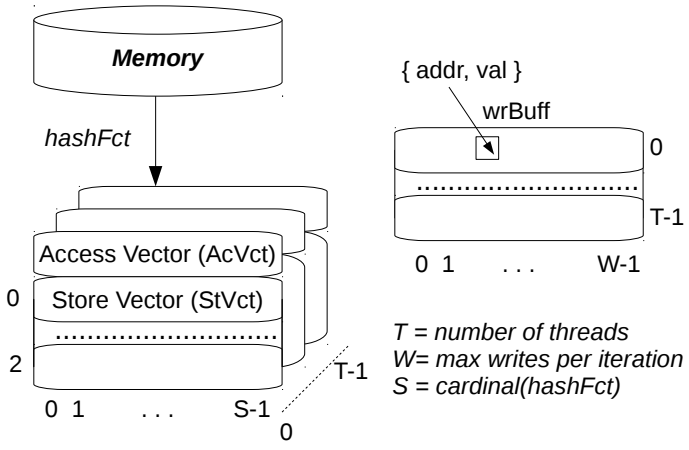


Figure 4.1: Speculative Storage Structure of SPLPC

Section 4.1 presents the structures used to monitor accesses to the protected memory areas. Section 4.2 describes the execution workflow of each iteration, with Section 4.3 describing the supported dependences. Section 4.4 describes the conflict detection mechanism, with an empirical evaluation in Section 4.5 and final conclusions in Section 4.6.

4.1 Speculative Storage Structure

SPLPC is implemented atop the POLYLIBTLS speculation library using SPLSC as a base. The speculative storage structure is detailed in Figure 4.1.

Each executing thread maintains its own private access vector (`AcVct`) and several store vectors (`StVct`). `AcVct` is a bitset that records the locations of all reads and writes the iteration makes to the protected memory areas. `StVct` is an array of three bitsets that records just the writes of the iteration. Each `StVct` array is used in a round-robin fashion, one per executing block of iterations. As with SPLSC and SPLIP, this new scheme employs hashing of memory addresses to track accesses to each location, vastly reducing speculative storage size. This does however allow for *false dependences*. The usage of `AcVct` and `StVct` is detailed further in Section 4.4.

Finally, `wrBuff` is a set of buffers, one per thread, recording the {`address`,`value`} pair of each write to the speculative area, ready to be committed to memory.

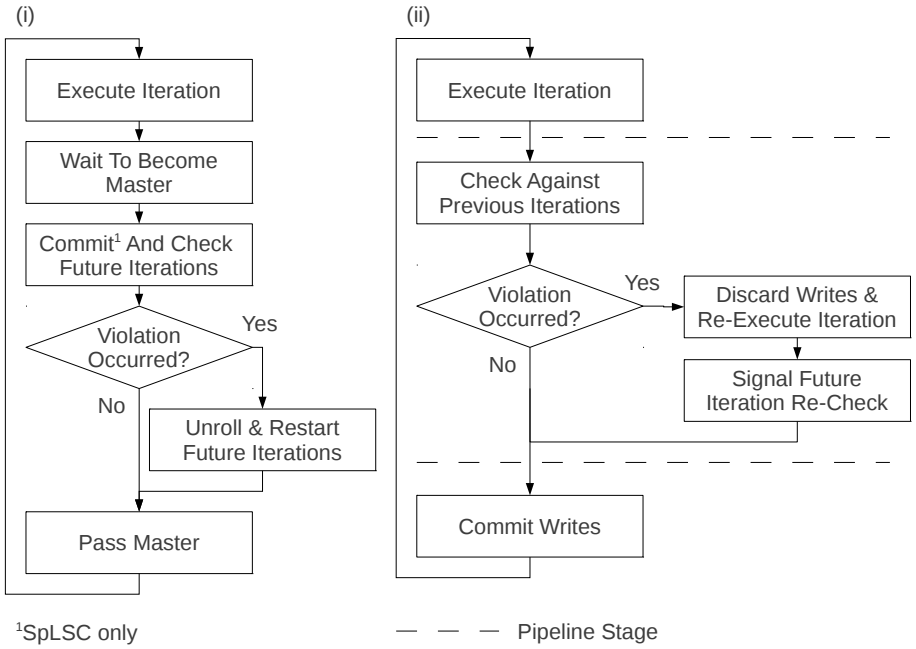


Figure 4.2: Speculative Workflow for (i) SPLSC and SpLIP, (ii) SPLPC

4.2 Pipeline Stages and Execution Workflow

There are two conditions that must be maintained for correct operation:

(i) Future iterations must delay checking for dependency violations until all previous iterations have finished executing, and

(ii) An iteration must not commit its writes until the current and all previous iterations have successfully detected no violations.

To that end, this design executes in a pipelined fashion preventing iterations from violating these conditions. Pipeline stages are implemented using two shared variables, each containing the index of the lowest-numbered iteration that has not finished executing or checking respectively, instead of using a single Master variable.

The execution workflow with a comparison to SPLSC and SPLIP is displayed in Figure 4.2. Threads execute entirely in parallel, but they are not allowed to progress to the next stage unless all previous iterations have done so. Under ideal situations pipeline stalls do not occur, and associated overheads account for a nominal setup time only. In the first stage, the thread executes storing any writes to `wrBuff`. Next, the thread checks for violations (as detailed in Section 4.4). Assuming none are detected, the thread progresses to the next stage: committing all writes back to memory. The threads do not

have to ensure that previous iterations have finished committing before proceeding to execute the next iteration. Should a dependence violation be detected, the iteration discards its writes and re-executes notifying all future iterations already checking that they must re-check for dependences. A single MFENCE instruction is used at the end of the commit phase to ensure that the writes become visible to all other threads.

4.3 Supported Dependences

By nature of speculative execution, all dependences that can occur are accounted for. Those that can occur and how they are managed can be split into two categories:

(i) **Non–loop-carried dependences.** Each thread's local writes are handled by the POLYLIBTLS library using the same methods as SPLSC. By performing writes sequentially for each iteration WAR and WAW dependences are handled automatically. RAW dependences are detected locally using a hashed bitset to determine if a write has occurred, and a linear search to retrieve values. This can be converted to a hash-based storage structure should RAW dependences be common.

(ii) **Loop-carried dependences.** Dependences that can occur between threads are tracked by the speculation model. The pipeline described in Section 4.2 enforces that any writes for an iteration are committed to memory only after previous iterations have finished executing; so all WAR dependences are automatically protected. The conflict detection mechanism described in Section 4.4 detects all other cross-thread dependences, triggering a rollback should a conflict occur. Correct execution is ensured, though there is a performance hit associated with this approach.

4.4 Conflict Detection

Detection of dependence violations is performed in a lazy manner. Dependences are detected by comparing the values of the thread-local `AcVct` to the `StVct`s of previous overlapping iterations. Should a previous iteration write to a location in memory that is read by the current iteration (in the case of RAW) or written to by it (in the case of WAW), then the corresponding locations in both `AcVct` and `StVct` will have been

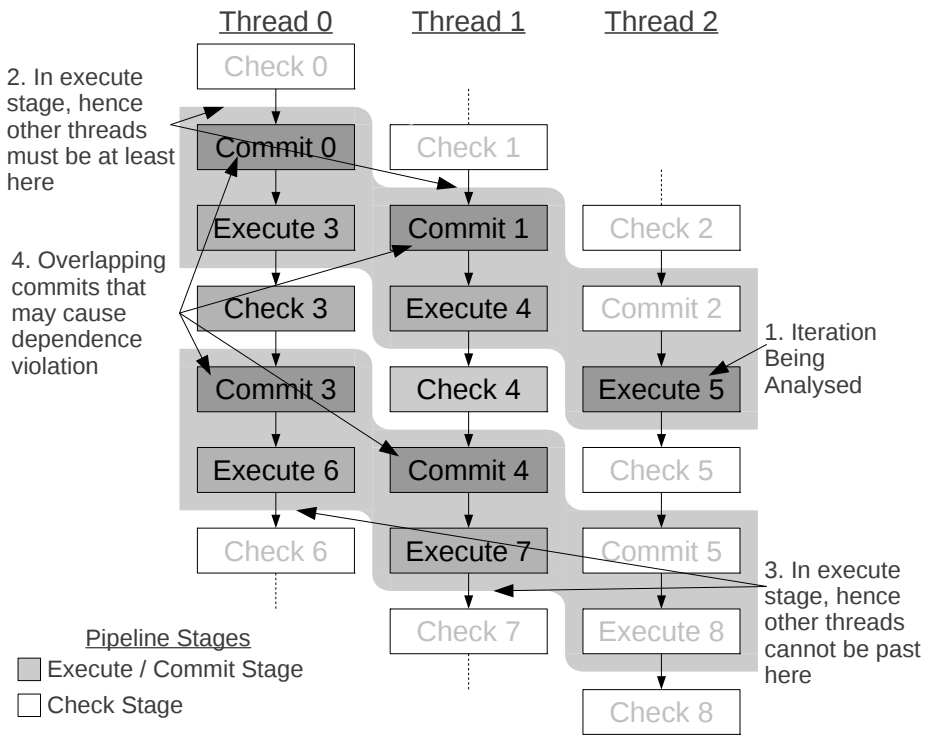


Figure 4.3: Possible Overlapping States At Iteration Execution.

marked, indicating a possible violation has occurred and thus triggering a rollback. Due to the hashing of memory addresses, false sharing between vector entries may occur causing unnecessary rollbacks; however, this conservative approach is preferred over incorrect execution.

As demonstrated in Figure 4.3, this comparison must be performed on the `StVcts` belonging to the previous two iterations from all other threads executing. Take for example iteration 5 in thread 2. At the time of execution, thread 0 can be at any point between committing iteration 0 and executing iteration 6. This indicates that any writes for iterations 0 and 3 may not yet have been committed back to memory and could interfere with the outcome of iteration 5. The same goes for thread 1, with iterations 1 and 4, and for any other previous two iterations of a thread should more exist.

The pipeline also enforces ordering between accesses to each thread's `StVcts`. Similar to the execution overlap discussed above, at any stage where a thread is able to write to a particular `StVct` it is guaranteed that the other threads will only ever be attempting to read from the other two vectors. If they need to check against the vector

being written to, the thread will block until it becomes available. This is not the case in general use, but the pipeline must be there to enforce consistency nonetheless. Similarly, the thread will not be able to write to the next `StVct` until the previous threads have finished checking against it. To that end, `StVct` contains three vectors accessed in a *round-robin* fashion: each has only a single writer or multiple readers, and never both. This method also helps reduce communication overheads by removing multiple-reader/multiple-writer volatile variables.

4.5 Empirical Evaluation

This section demonstrates that this new model works well on loops exhibiting regular patterns, and does so with performance comparable to that of the other models. First is a description of the testing and data-collection methodology, followed by a presentation and analysis of the results. All experiments were executed on a machine with four dual-core AMD Opteron processors, model 1218 at 2.6 GHz, with 16 GB of RAM memory, running Red Hat Enterprise Linux 5. Each benchmark was compiled using GCC 4.1.2 with `-O3` optimisation and the *pthread* library.

4.5.1 Experimental Methodology

For evaluation of the speculation scheme, a number of experiments were executed using four industry-standard benchmarks selected from the NAS Parallel, Bytemark and Scimark2 Benchmark Suites. The benchmarks were chosen due to the abundance of parallelism available in the *loop-kernels* they contain. They were also selected due to their heavy use of pointer arithmetic, which makes them difficult to parallelise using only static information. As a baseline, only the execution times of those loops parallelised have been considered, excluding any setup and teardown times associated with the benchmarks. For benchmarks containing several suitable loops, the loops selected are structurally very dissimilar, with each warranting individual performance analysis. These loops were extracted and tested separately.

Additionally, each loop has been provided with a range of input data sets to show how the performance of speculative execution can vary on identically structured loops. This is due to varying overheads associated with speculative models, attempting to

determine under what circumstances each model will overcome these overheads. The range of workloads provided to each loop was chosen based on reasonable limits that the loop would encounter, varying to a different degree for each loop. With every workload each benchmark executed in the range of a few seconds to minutes.

To allow for a direct comparison with other techniques, the benchmarks were selected based on those used by other researchers [29]. The benchmarks used are as follows:

(i) EP (Embarrassingly Parallel) benchmark from the NAS PARALLEL BENCHMARK suite. The main loop of this benchmark, used for generating random numbers, involves no communication between threads.

(ii) IDEA from the BYTEMARK suite contains two main loops, CIPHER and DEKEY referring to the encryption and decryption sections of the algorithm.

(iii) NEURALNET also from BYTEMARK contains many loops which can be generalised into two patterns of execution: NNETFW processing the forward pass on the output layer, and NNETBW adjusting the weights for the output layer.

(iv) SPARSEMATMULT from the SCIMARK2 suite, which implements multiplication across a large but sparse matrix.

All experiments were performed both sequentially and speculatively using the available SPLSC, SPLIP and SPLPC models. For each model, a range of possible combinations of parameters were selected, with the thread count being 2, 4 or 8, and the unroll factor tested at 30 different data points between 1 and 16384. The hashing algorithm for dependency detection was fixed at a size large enough to avoid a rollback.

4.5.2 Summary of Key Results

A comparison of the best speedups for each model and workload is shown in Figure 4.4. On average the pipelined commit model performed 15% slower than the serial commit model, but was able to execute up to 95% faster. This was due to the novel pipelined commit phase, preventing some loops from being overwhelmed by locks and critical sections. Due to this and the memory management it is also more scalable than the serial model.

Compared to the in-place model the new design executed on average 28% faster, and up to 143% faster. This is thanks to the differences in memory management and handling

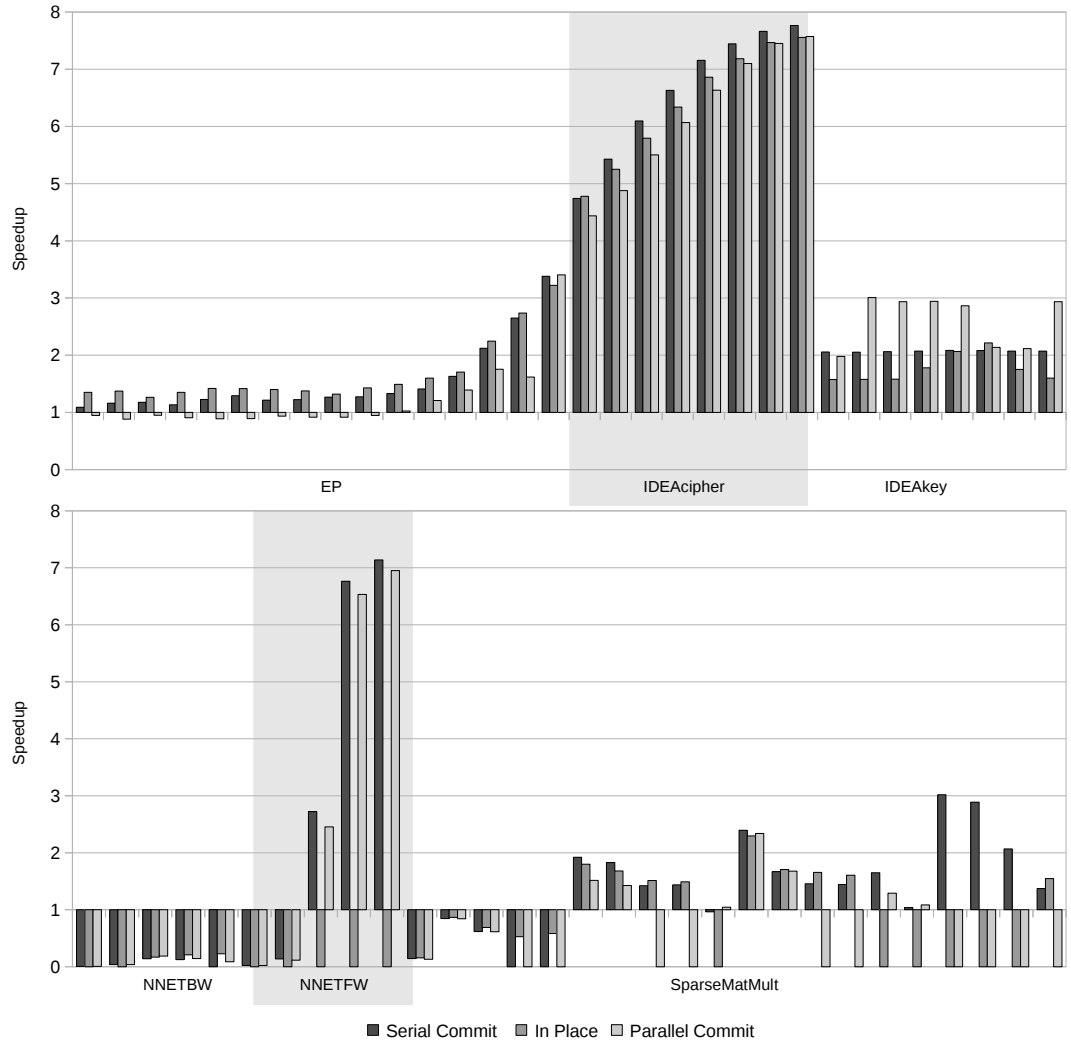


Figure 4.4: Speedup for each loop using the best seen speculation policy. Each result represents a separate input provided to the loop, ascending in size of workload.

of locks and shared variables that can cause the in-place model to be bogged down by administrative overheads.

It is notable that in the best case, speculation achieves the highest performance for the largest data sets. This is due to the large overheads associated with setup of the speculative data structures, and the ability to set the unroll factor high enough to hide any overheads associated with the dependence check/commit phases.

As can be seen each model performs better for some workloads than others. In many cases the use of the wrong model will result in a significant slowdown of execution. For SPLPC this is generally the case with smaller workloads, where the setup and teardown costs have more of an influence on the overall speed of execution. Addi-

tionally for some executions the added synchronisation costs associated with having multiple pipeline stages cannot be hidden among iteration execution times. A solution to this would be to increase the loop unroll factor further, however with a low number of iterations this may not be possible or in doing so the rollback buffer may become large and unweildy, bogging down performance further. This is a tradeoff that must be accounted for and, in some situations, require the use of an alternative speculation scheme.

4.6 Conclusion

The evaluation of the novel Pipelined Commit model found it provided consistent speedups on par with the serial commit and in-place models tested. It is clear from this work that there is no "one size fits all" speculation scheme, and that the Pipelined Commit model is suitable for supplementing existing schemes to extract additional performance.

It is also clear from the evaluation that the correct selection of a suitable speculation scheme is extremely important to the performance that can be achieved. Additionally, the selection of correct parameters for each model is equally important, and a difficult task to perform manually. The next chapter introduces a machine-learning–based selector for models and policies, allowing this process to be automated.

Chapter 5

Smart Speculation Policy Selection

Analysis has shown that for SW-TLS, there is no "one size fits all" model for each program that could be encountered [29]. Thanks to the flexibility that software implementations provide, there are a large number of SW-TLS models available. The performance of each model varies greatly depending on the structure of the code that is being executed, in many circumstances being detrimental to the performance compared to sequential speed. Therefore, it is important to choose the correct model and model parameters to ensure a greater performance and, where necessary, run the loop sequentially.

When considering auto-parallelisers that employ SW-TLS, it is especially important that this decision be automatable. As such, this chapter presents and evaluates a machine-learning-based system for automatically selecting an appropriate speculation policy. This chapter initially discusses a motivating example in Section 5.1. A discussion of various factors that can affect the performance of a given speculative loop is in Section 5.2.1, and a summary of the parameters that are commonly found in speculation schemes can be found in Section 5.2.2. The design and workflow of the policy selector is described in Section 5.3 with a thorough empirical evaluation presented in Section 5.4. Finally a summary of key results is found in Section 5.5 and final conclusions in Section 5.6.

5.1 Motivating Example

Consider the loop in Figure 5.1. A cursory glance will show that this loop can be

```
for i = 1 → outer_total do
  for j = 1 → inner_total do
    local[j] ← speculative[i][j]
  end for
  do_something_local(local, run_time)
  for j = 1 → inner_total do
    speculative[i][j] ← local[j]
  end for
end for
```

Figure 5.1: Motivating Example for Automated Policy Selection

safely parallelised on the outer loop without the need for speculation, but assume that the compiler is unable to determine this and hence reverts to speculation. Depending upon the speculative model being used this program's performance will react very differently.

With many models, should the outer loop have too few iterations (low $outer\_total$) all execution of the loop will be dominated by administrative overhead: that is, the setup of the speculative buffers and access structures. A requirement with all models is that there are sufficient iterations to overcome these overheads.

Next, consider values of $inner\_total$. This variable controls the number of speculative accesses for each iteration. If there are too many speculative writes then models with a serial commit phase are inappropriate as threads stall waiting to enter critical commit sections. In this case the selection of a model that makes writes directly to the protected area must be selected, however there are often greater memory and synchronisation overheads associated with such models. Should $inner\_total$ be low then selecting a model with a serial commit phase may be more appropriate, but if it is too low then commit overheads will dominate execution. Again, threads will be forced to wait. This can be overcome if there is sufficient computation on each outer iteration, in this example controlled by the parameter run_time.

Finally, the hardware that the code is executed on directly affects the performance. Differences in architecture and importantly memory bandwidth can have an effect, for example causing models with a large amount of shared variable interaction to be limited. Counter-intuitively, adding more hardware such as additional processors can have a detrimental effect on performance due to shared variables becoming thrashed,

or threads being held up by critical sections.

For example, take the NNETFW benchmark (discussed in section 5.4). Testing has shown that for large networks with many iterations speculation provides respectable speedups of up to 7.14. However, on smaller networks with very few iterations, speculation can cause execution to take over 50 times longer compared to sequential execution. Similarly, even on the large loop that generated a 7.14 speedup, a misconfigured speculation policy can cause execution of over 30 times as long. Hence it is extremely important that speculation, its models and their parameters are considered carefully to ensure a performance gain. Doing this manually for every loop is difficult and time-consuming, and completely rules out auto-parallelisation techniques. Hence the automated method proposed by this chapter is desirable for both manual coders and more aggressive auto-parallelisation tools.

5.2 SW-TLS Configuration

The correct configuration of SW-TLS schemes is a complex issue due to the high number of factors that can affect the performance of a program. These include, input factors such as the code that is being executed and the hardware that it is executed on, and the specific configuration of speculative execution.

In order to adequately employ automated parallelisation the input factors are generally fixed except for the use of code transformations, however speculative execution is intended for use when static analysis has failed making the use of code transformations all but impossible. These factors are discussed in Section 5.2.1.

Instead, there is much flexibility in the configuration of a speculative policy. Not least the selection of an appropriate speculation scheme, each scheme contains a number of configurable parameters. These parameters cover all sections of the speculative execution, including how and when memory traces are collected and analysed, how backup data is stored and discarded, and how speculative threads interact. It also includes a number of more complex settings, such as the granularity of address hashing techniques. All options are ripe for use with machine learning, however some, such as learning a hash function based on input data sets are more complex issues than others. The parameters used for training and predictions are discussed in Section 5.2.2.

5.2.1 Factors Affecting Performance

A number of speculation models have been analysed to extract various metrics that affect their performance. These metrics have been extracted through manual experimentation of existing speculation schemes and are used as inputs to the the machine learning based policy selector. The main *input features* that appear to affect a model's performance are:

(i) **Speculative accesses made per iteration and in total.** More speculative writes generally cause longer commit times and can cause worse performance if there are too many.

(ii) **Instructions executed per iteration and in total.** This gives a measure of how much computation the loop performs other than speculative accesses. Less computation can cause execution times to be dominated by speculation overheads.

(iii) **Total number of iterations to be executed.** There must be an adequate number of iterations for speculation setup and teardown overheads to be overcome.

(iv) **The ratio of instructions to speculative accesses.** This is another measure of the amount of time spent executing the loop compared to the amount of time spent in the speculation library.

5.2.2 Common Speculation Parameters

Many speculation models also include parameters to fine tune them and obtain the greatest performance. These can often include:

(i) **Thread Count**: dependent upon the loop being executed this can often be fewer than the number of processing units available. If other parameters are chosen poorly then threads may stall entering critical sections. Executing with more than the optimal number of threads introduces an unnecessary drain on memory bandwidth and other resources.

(ii) **Unroll Factor**: the number of iterations executed by each thread before expensive dependence checks and write commits are performed. By unrolling the loop, overheads associated with these sections can be minimised, but if this is set too large then write buffers can become unwieldy. This is one of the main causes

of performance degradation associated with speculation. A related metric is the ratio of unroll factor to total number of iterations.

(iii) **Access Vector Size**: the size of the speculative data structures used. With many models, the hash of each memory location is calculated and used as an index into the data structures. If the load vector size is set too low many memory locations will refer to the same index increasing the likelihood of a *false positive*. If this is set too large, then memory usage can become excessive and structure reset times can cause degraded performance.

(iv) **Buffer Size**: the size of the buffers used to store pre-commit or rollback information. In general this must be set to the maximum numbers of writes that can occur in an iteration or unrolled section.

These parameters are the outputs of the machine learning based policy selector.

5.3 Policy Selection Workflow

To ease the selection of an appropriate speculation scheme and its parameters for a given input program, this chapter proposes the use of a machine-learning-based speculation policy calculator. This section introduces the training methods used for the policy selector in Section 5.3.1, followed by the workflow used to perform the policy selection in Section 5.3.2.

5.3.1 Prediction Model Training

The policy selector must undergo an initial *offline* training period. A wide range of loops, including those both suitable and unsuitable for speculation, must be executed with their runtimes recorded. They are initially executed sequentially, and then through each available model across a subset of the available parameters, at which point the speedup of each execution is calculated. The predictor is then trained using only the best performing speculation policy for that loop, using the loop characteristics described in Section 5.2.1 as inputs, and the speculation parameters described in Section 5.2.2 as outputs. This initial training period can be expensive and must be redone for new architectures, but this process is one-off and can be fully automated. The time taken to build or reload the predictive models after training is nominal.

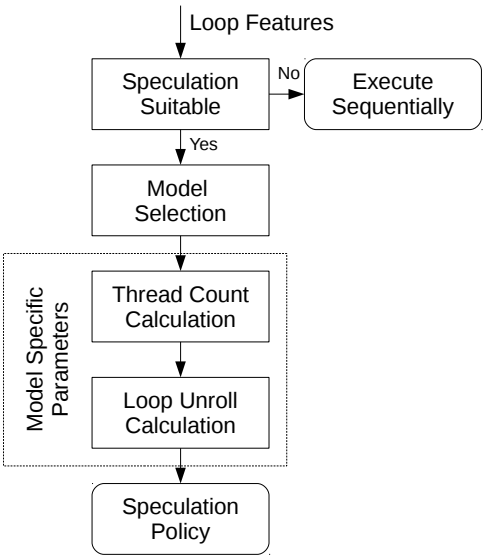


Figure 5.2: Prediction Workflow of Policy Selector

5.3.2 Policy Calculation

Figure 5.2 details the sequence of predictions made to calculate a suitable speculation policy. Training data for each stage is selected based on the predictions made in the previous stages, to ensure only relevant data points are used. The decision flow proceeds as follows:

(i) Is the code suitable for parallelisation using speculation? If this prediction fails, then speculation will likely slow down execution and should not be used.

(ii) Which model is most suitable? Often several models will provide speedup, however some will outperform others.

(iii) Model-specific parameters, the number and type of decisions made at this stage can vary, however they are identical for models evaluated. These are thread count and loop unroll factor as described in Section 5.2.2. Calculation of the speculative storage size depends on more complex memory access patterns and warrants individual research beyond the scope of this work. In each case it has been fixed large enough to not cause a false positive.

5.4 Empirical Evaluation

This section demonstrates that the proposed machine-learning technique provides good accuracy at deciding if a loop is suitable for speculation, and at selecting a speculation policy that will provide consistent performance gains if a loop is so suitable.

First the testing and data-collection methodology is described, followed by a presentation and analysis of the results. All experiments have been executed on a four socket dual-core AMD Opteron machine, model 1218 at 2.6 GHz, with 16 GB of RAM memory, running Red Hat Enterprise Linux 5. Each benchmark was compiled using gcc-4.1.2 with -O3 optimisation and the *pthread* library.

5.4.1 Evaluation Methodology

For training and testing of the policy selector, a number of experiments were run using the same 4 industry standard benchmarks as described in Chapter 4. The benchmarks were chosen due to the abundance of parallelism available in the *loop-kernels* they contain. They were also selected due to their heavy use of pointer arithmetic, making them difficult to parallelise with only static information. As a baseline, only the execution times of those loops parallelised have been considered, excluding any setup and teardown times associated with the benchmarks. For benchmarks containing several suitable loops, the loops selected are structurally very dissimilar, with each warranting individual model and parameter selection. These loops have been extracted and tested separately.

The loops have been configured and executed in the same fashion as Chapter 4 with the only exception that if an execution took significantly longer than the sequential time (greater than 5 times) then it was cancelled as it would not provide any useful insight into the performance of the model. These parameter selections provide a wide base of training data to depict how the different models perform under each condition.

5.4.2 Policy Selection Testing

To evaluate the policy calculator, the new method was applied to every benchmark setup we have collected data on. At each stage the predictor was trained in a *leave-one-out* fashion at the granularity of the loop being parallelised. That is, for each loop

Method	Parameter	Value
K-Nearest Neighbour	Neighbours	1 - 3
Naive Bayes	No parameters	
Neural Net	Training Cycles	500
	Starting Weight Change	0.3
	Incremental Weight Change	0.2
Support Vector Machine	Kernel Type	Dot Product
Linear Regression	No parameters	

Table 5.1: Summary of Machine Learning Parameters

tested, every data point for every workload associated with that loop was removed from the training data. Every data point for that loop was then tested and the speculation policy and speedup obtained was stored. This process was repeated for every loop tested. Using this method allowed estimations of how the policy calculator will perform when given a completely unseen loop. The speedup obtained using the calculated policy was compared to the best seen configuration for that loop. A comparison between calculating the unroll factor and the ratio between unroll to iterations was also performed to see which method provided more useful results.

5.4.2.1 Machine Learning Techniques

Each predictor was trained and evaluated using the RAPIDMINER [26] software package, which allows easy testing of a number of different machine learning techniques. The decisions *whether to speculate*, *which model to use* and *how many processors* were treated as discrete decisions, with the *unroll value* and *unroll ratio* calculated as *unbounded* or *continuous* values respectively. The unroll value was then rounded to the nearest whole value.

For any discrete decision, Decision Tree, K-Nearest Neighbour, Naive Bayes and Neural-Net-based deciders were evaluated. For unbounded/continuous decisions K-Nearest Neighbour and Neural-Net deciders, Support Vector Machines and Linear Regression Techniques were evaluated. Descriptions and analyses of these methods can be found in [16]. Table 5.1 states the chosen machine-learning parameters.

Each stage of the predictor was tested separately, and compared to the best possible

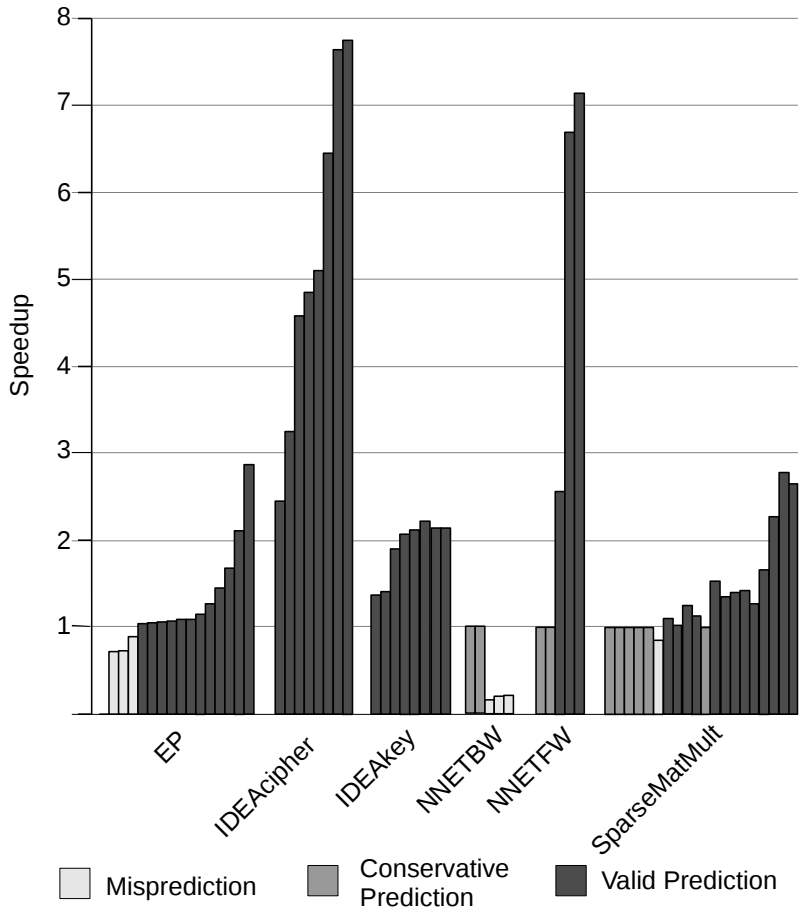


Figure 5.3: Speedup for each loop using calculated speculation policy on an 8-core machine. Each bar represents a separate input provided to the loop, ascending in size of workload.

combination that was witnessed during data collection. Future stages of the predictor used the best set of predictions from the previous stage as their input, and after the last stage the speedup of the policy was calculated. If the policy had not been seen during data collection then it was executed again, but the secondary executions were not used as training data.

5.5 Summary of Key Results

Figure 5.3 gives the results for the best set of policies calculated, with Figure 5.4 displaying the best speedups achieved by each model. The policies calculated provide speedup with a geometric mean of 1.64 and a maximum of 7.75 times that of sequential

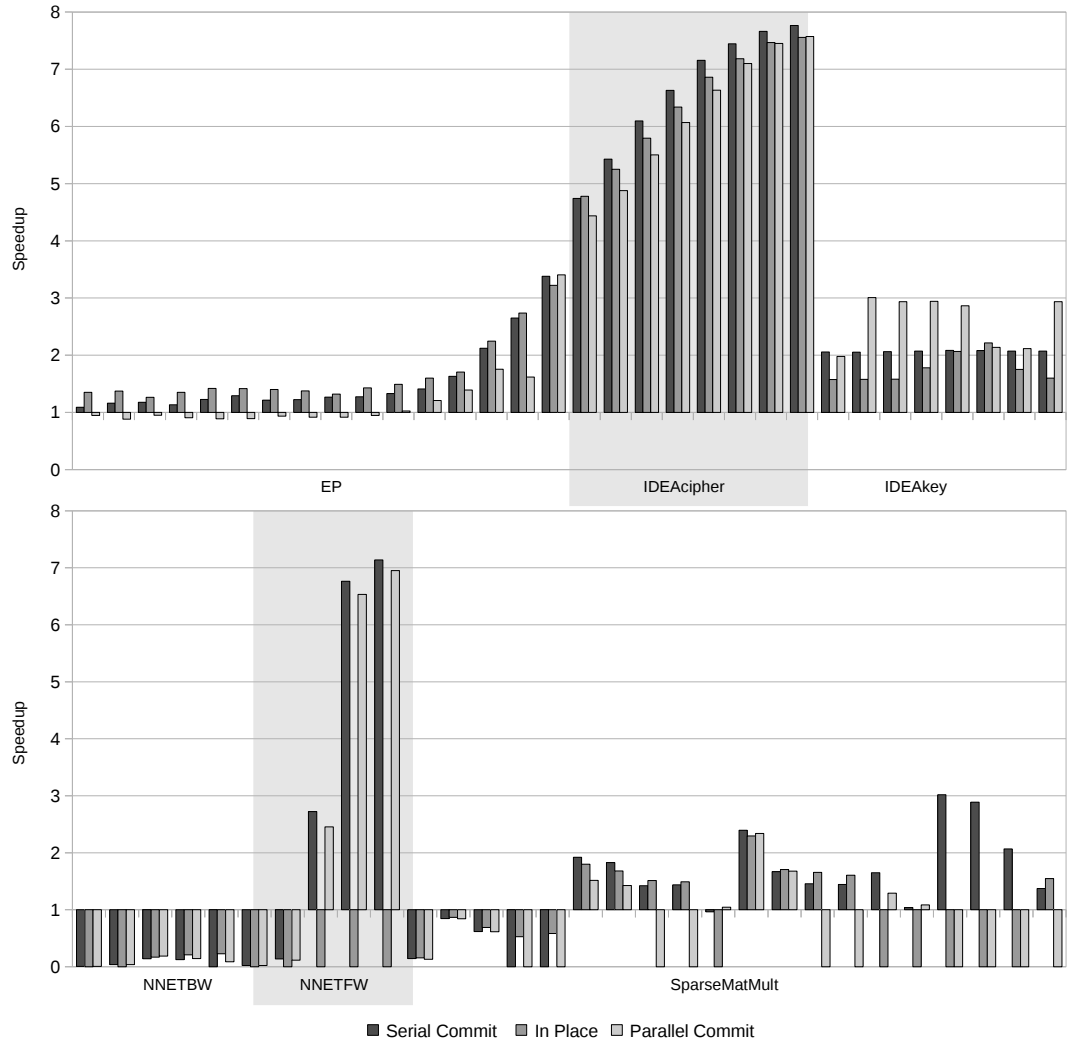


Figure 5.4: Speedup for each loop using the best seen speculation policy on an 8-core machine. Each result represents a separate input provided to the loop, ascending in size of workload.

execution. 89% of the policies resulted in a speed of sequential or better, however there are 7 policies that resulted in a direct slowdown, 3 of which were from incorrect predictions in the first stage of the calculation, with the other 4 coming from poor choice of model and/or model parameters. Of the conservative predictions, only one was miscalculated, where the speedup witnessed during data collection had been just 1.04.

The first stage of the predictor, whether or not to use speculation, is arguably the most important as for many pieces of code speculation is simply not suitable and causes slowdown. As shown in Figure 5.5 it is possible for this predictor to be accurate

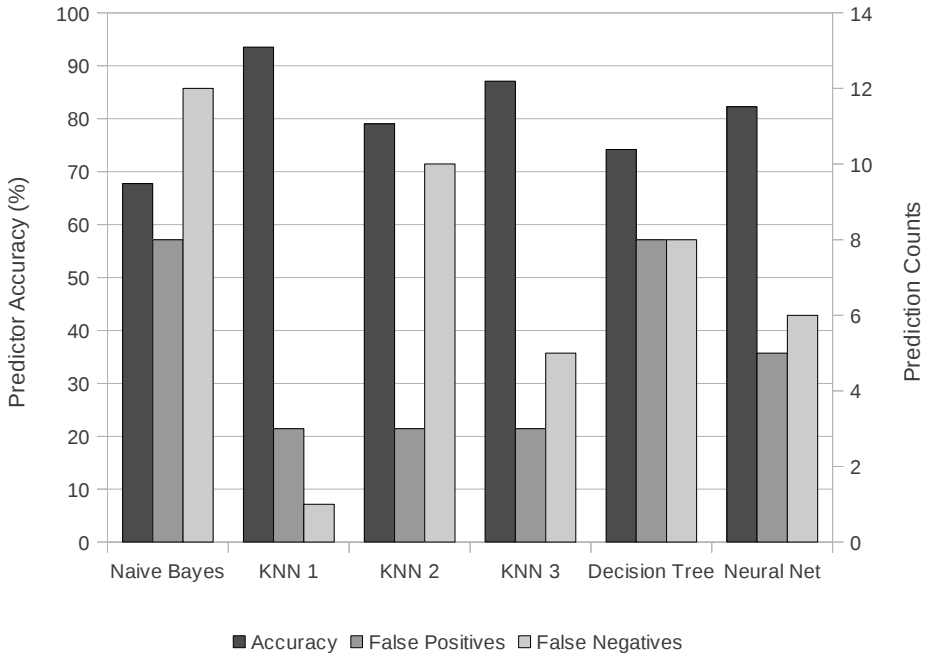


Figure 5.5: Accuracy of *Speculation Suitable* predictor (Left axis), and the error types (Right axis)

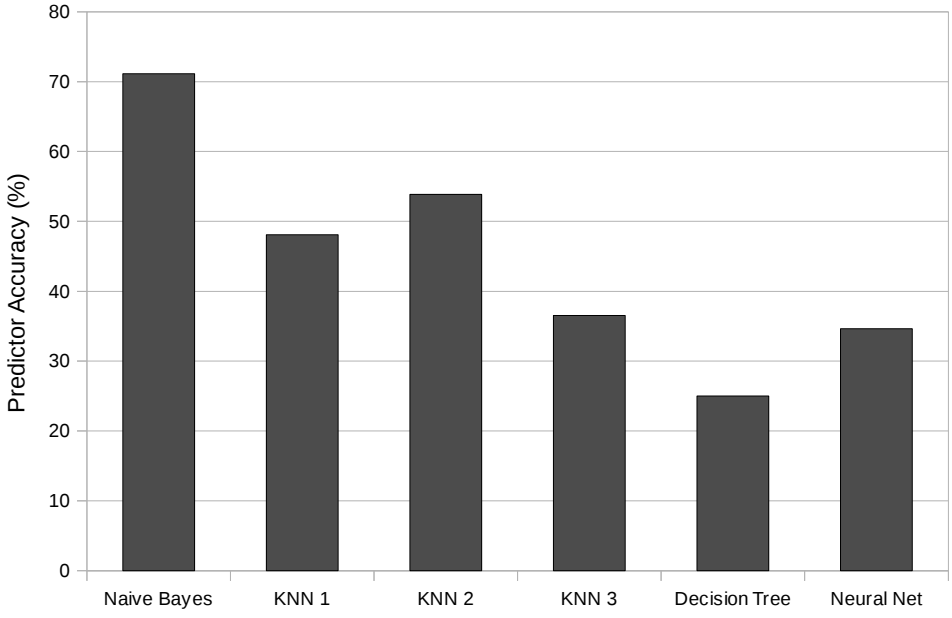


Figure 5.6: Percentage of queries that yield the best model selection as seen during data collection

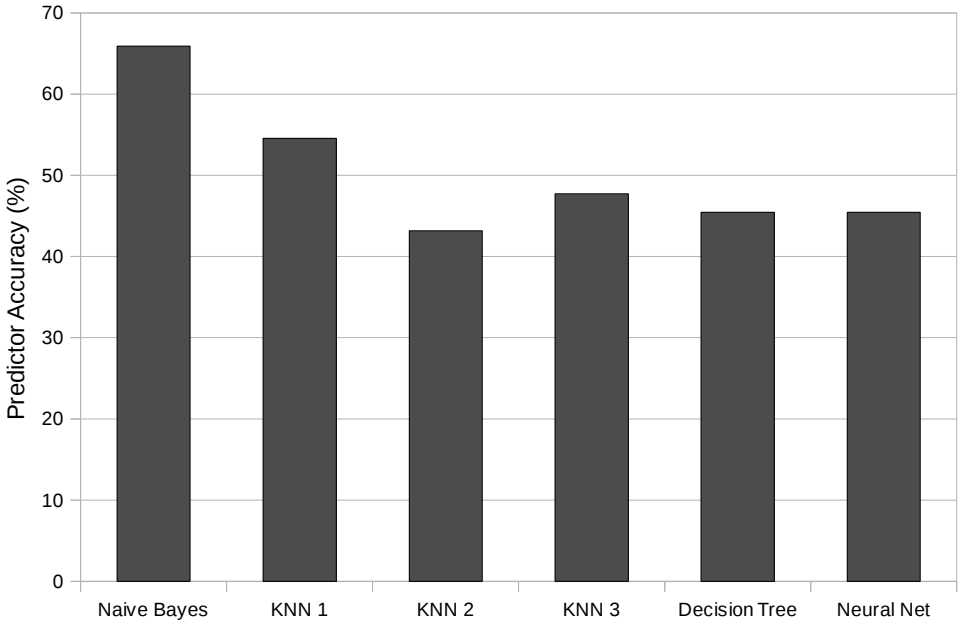


Figure 5.7: Percentage of queries that yield the best thread count selection as seen during data collection

when provided with suitable training data. Testing showed that using a K-Nearest Neighbour predictor at this stage with k=1 yielded the best results, with an accuracy of 93.55%. Of the 62 predictions made three were falsely detected as speculative and are likely to execute slower, and just one false negative, when speculation would have been appropriate.

As can be seen in Figure 5.6 the second stage, which model to use, was less accurate, with the best option selecting the best model 71.15% of the time. However, this result is based around the best model that had been seen, with other models and policies still able to provide a speedup. A simple naive-bayes-based selector was the most accurate, with other machine learning methods performing much worse.

Similarly, the thread count selection was less accurate again, with the best technique selecting the most suitable value 65.91% of the time. Figure 5.7 displays the accuracies calculated, with a simple naive-bayes- based selector performing best.

The final stage of the predictor cannot easily be compared with the best option due to the unroll factor being unbounded. Without exhaustive testing that would take an excessive amount of time the choice of unroll can only be compared to the speedup the chosen policy provides, compared to the best seen during data collection. Table 5.2 gives the geometric mean and maximum speedups obtained across the various machine

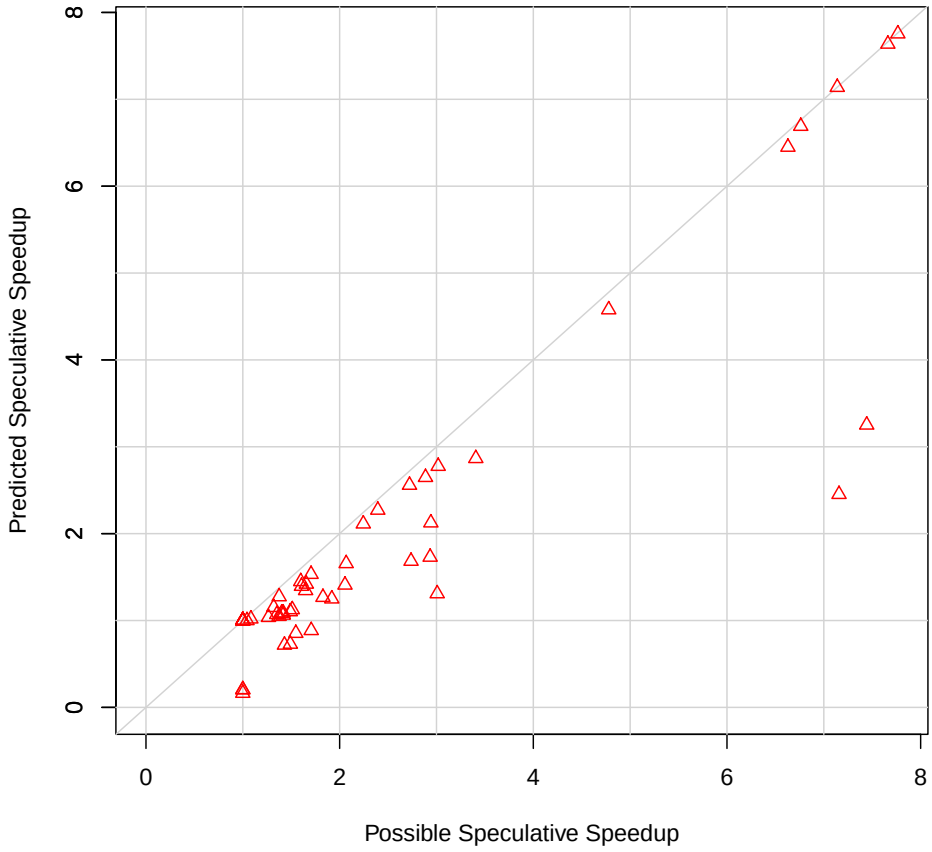


Figure 5.8: Speedup Using Calculated Speculation Policy Versus Best Previously Seen Speedup

learning techniques tested. As can be seen K-Nearest neighbour with k=2 performs best with a geometric mean speedup of 1.64 and up to 7.75. Figure 5.8 shows the comparison of the speedup obtained using our method, with the best speedup obtained during data collection. On average this method was able to obtain 74% of the speed seen during training. Calculating the ratio of the unroll factor to the total number of iterations proved to be the best method as simply calculating the unroll-factor directly often produced some unroll values of several times the actual number of iterations.

As mentioned the accuracy of results decreases with each stage of the predictor. This can be partially accounted for by the decreasing number of options available at each stage and the difficulty of each decision. For instance, the first stage, whether or not speculation is appropriate, is a simple boolean decision that can be estimated more accurately using higher level statistics. At this stage there are many different models

	Unroll		Unroll Ratio	
	Geomean Speedup	Maximum Speedup	Geomean Speedup	Maximum Speedup
KNN 1	1.28	7.71	1.63	7.75
KNN 2	1.39	5.7	**1.64**	**7.75**
KNN 3	1.33	3.02	1.64	7.59
NNET	1.22	5.34	1.72	5.92
Linear Regression	1.24	2.91	1.38	6.21
SVM	1.32	6.06	1.47	7.63

Table 5.2: Speedups Using Calculated Speculation Policy Of Different Unroll Prediction Techniques

and other configuration parameters that may provide a speedup, therefore for many loops it is likely that if it satisfies certain conditions then there will be a speculation scheme that will provide speedup.

In contrast, the selection of which scheme will provide the best speedup is a more difficult decision, partially because multiple schemes may provide a satisfiable speedup, and that the exact conditions under which it will perform well may require more information than those empirically discovered in Section 5.2.1, similar to how the selection of an appropriate hash function requires a more advanced knowledge of the memory access patterns of the loop. However, this chapter demonstrates that despite lower accuracy in each stage, the selection of a reasonably well-performing policy can still be acheived with this high level knowledge. Further research into the loop and scheme analysis will likely result in more accurate results and a better performing speculation policy.

5.6 Conclusions

This chapter has presented an integrated approach to efficiently implementing software TLS on machines without architectural TLS support. Observing that there is no *one-size-fits-all* solution in software TLS, this chapter has devised a machine-learning-based technique to automatically select a suitable software TLS model and its detailed configuration. The results obtained on a standard 8-core AMD machine demonstrate

that software-only TLS can deliver consistent speedups of up to 7.75, and with a geometric mean of 1.64, across compute-intensive and difficult-to-parallelise benchmarks.

In general, simpler machine learning techniques perform best. For discrete choices such as model selection and processor counts, naive bayes provides the best speedups. For unbounded/continuous decisions, such as the unroll ratio, a nearest-neighbour decider using only one or two neighbours worked best. Additionally for unbounded decisions, calculating the ratio with respect to the number of iterations works best.

This novel technique is ideally suited to complement profile-based parallelisation approaches such as [44]. These techniques never execute known sequential loops as parallel, and any remaining loops have a very low probability of dependence violations occurring.

Chapter 6

Automated Error Checking for Aggressive High-Confidence Parallelisation

This chapter explores the hypothesis that standard, heavyweight TLS methods are often not required for aggressive parallelisation. With the advent of modern static analysis and profiling techniques [44] it can be demonstrated that a loop or section of code is unlikely to contain a data dependence with very high confidence. In these circumstances it may be more appropriate to remove the ability to checkpoint and rollback and choose a very lightweight error-checking scheme over a full TLS system. This would allow for faster and scalable execution at the risk of an extremely high re-execution cost in the unlikely event that a dependence violation is detected.

The architecture of a lightweight error-checking scheme can be similar to that of a standard TLS scheme; the most significant difference is that version control is no longer required. Memory traces are still recorded during execution and used to detect dependences, but each speculative access need no longer be backed up or committed to memory at a later time. Instead, when a dependence is detected the entire process must be killed and re-executed in a safe, non-speculative manner. This significantly reduces the memory footprint as backup/commit logs are no longer stored. It also eliminates the computation and synchronisation between threads on each access that would otherwise be needed to maintain those logs. This can also allow further optimisation of memory-trace techniques.

The goal of this work is to remove as much of the tracking and dependence overhead from the critical path of execution as possible, while still being able to provide a guarantee of correct execution. This introduces three main performance considerations:

(i) **Computation.** The amount of work executed for each speculative access has a direct impact on the critical path of execution. By removing safety from TLS, computation is significantly decreased; however, memory trace and dependence detection methods must still minimise computation.

(ii) **Synchronisation.** One of the largest obstacles to TLS scalability is how frequently each thread must interact with others, be it by sharing data or having to wait at barriers. By minimising the number of points during speculation where threads must interact, the critical path of each thread is less interrupted; this allows for a faster and more scalable TLS scheme.

(iii) **Work Distribution.** Similar to synchronisation, all computation should be distributable evenly between each thread. This is to limit a scaling bottleneck where additional threads increase the load on a single thread.

With these performance considerations the most suitable scheme is one with an extremely lazy checking schedule and an entirely distributed trace topology to remove as much cross-thread synchronisation as possible.

Section 6.1 briefly discusses the types and limitations of parallelism that this work targets. Section 6.2 presents the customised data structure used to store memory traces, with analysis of how it limits memory usage and computation. Sections 6.3 to 6.5 present several methods to perform dependence checks and analyses how each method distributes work between threads. Section 6.6 discusses the automated code transformations required to enable error-checked parallel execution, with several static analysis methods that can be performed to optimise the resulting code. Section 6.7 provides a thorough empirical evaluation, with final conclusions in Section 6.8.

6.1 Parallelisation Target

There are many forms of parallelism to which this style of TLS would apply. To limit the scope required for evaluation, this has been restricted to simple DOALL loops. More specifically: standard C/C++ `for` loops in which every iteration is guaranteed

to execute, such as those that do not contain `break` or `goto` statements. These loops can include nested loops, for which the restriction of `break` statements does not apply; however any `goto` statements contained cannot jump to beyond the outer loop.

As a further restriction this technique cannot be used on a program that performs irreversible I/O. If the program modifies another file, sends network traffic, or even has its output piped to another program then any I/O performed in a speculatively-executed region cannot be undone without additional support. In contrast, it is safe to use this technique if the program simply reads in information from an immutable file, as a re-execution would be able to read the same data.

In this work, parallelisable loops were selected using the profiling techniques presented in [44]. Discovered probably-parallel loops were marked up using the *OpenMP* parallel framework [3], with privatised and reduction variables added as appropriate. Further analysis and code transformations required for speculative execution are discussed in Section 6.6.

This technique can be used on other forms of parallelism, such as parallel threads. In parallel schemes using simple synchronisation, such as barriers, memory traces would be recorded between synchronisation points, and dependence checks triggered at each synchronisation point. For programs using more complex synchronisation (such as some producer/consumer models or pipelined threading) the checking schemes presented in this work would have to be modified, similar to those used by SPLSC [29], SPLIP [28] or SPLPC (Decribed in Chapter 4).

6.2 Memory Trace Data Structure

For any TLS system it is important to use a compact and efficient data structure to record memory access traces. Unfortunately, using a distributed, extremely-lazy tracking scheme also creates the largest memory traces. Further, standard speculation techniques for lowering the memory footprint, such as those discussed in section 2.3.6, are unsuitable for use as they introduce the possible detection of false dependences. In this design the existence of false dependences introduces too high a risk of triggering an extremely expensive re-execution of the entire program. Therefore the design of the memory trace structure is centred around the following tenets:

(i) It should be impossible for distinct memory addresses to be recorded as identical.

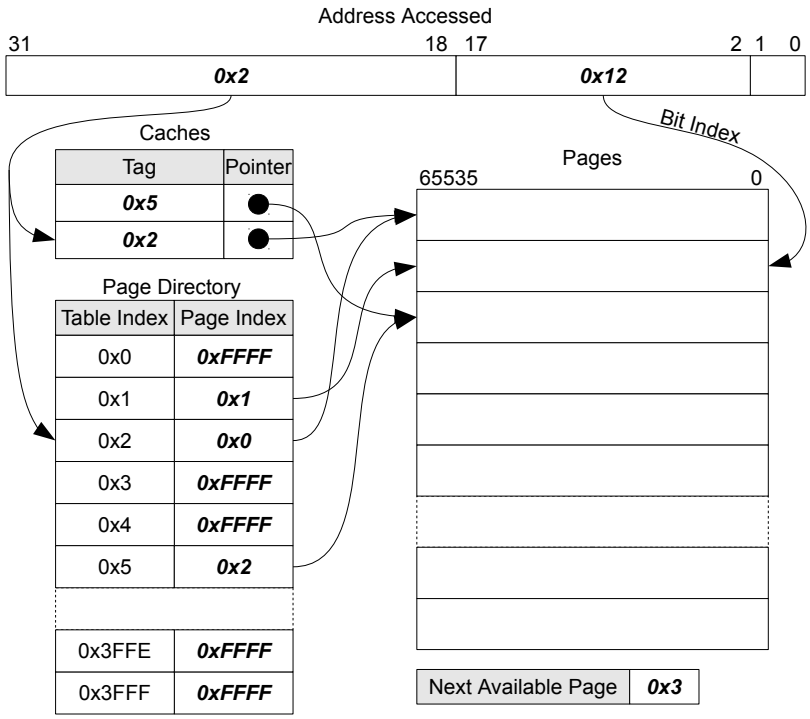


Figure 6.1: Page Table Structure

(ii) Memory should be allocated to each trace only for regions that are actually accessed.

(iii) Both temporal and spatial locality of speculative accesses should be taken advantage of.

(iv) The structure should be able to grow as necessary to handle as many accesses as possible, across any memory range.

(v) The memory allocated for each address accessed should be as low as possible — optimally only a single bit per address accessed.

Following these tenets the most suitable data structure for memory traces is similar to that of a page table as used in OS memory management systems. As shown in Figure 6.1, the structure contains a directory and a pool of available pages. Each page in the pool is a bitset, implemented as an array of unsigned long integers, with each bit representing whether a particular address has been accessed. Each directory entry contains either an index into the pool of pages for which that entry has been assigned, or a special value to indicate that that entry has not yet been assigned a page. On each access, the address of the access is split into two sections. The upper section is the index into the page directory used to determine which page is tracking

that set of addresses. The lower section is the bit index used to set the relevant bit within the page to show that it has been accessed. In the evaluation of this work, word-aligned tracking was required, so the two least-significant bits are discarded. A counter is maintained for which pages have been assigned, and incremented every time an unassigned directory index is accessed, with the previous value stored in the directory for that index.

The first design tenet is satisfied as every address is uniquely storable in the table. The second tenet is satisfied as each page is only allocated/assigned when it is used. The third tenet is also satisfied as the assignment of a page to a range of memory addresses exploits spatial locality: accesses to one address are likely to be followed by subsequent accesses in the same memory region. A further optimisation to exploit temporal locality is discussed in Section 6.2.1. The fourth tenet is satisfiable by ensuring there is a large enough and expandable pool of pages to be assigned, and ensuring that the directory entry element is large enough to index all available pages. The use of a pool of available pages lowers allocation and initialisation costs for each speculative section. Finally the fifth tenet is satisfied based on the efficiency of usage of a page. The number of bits used per address accessed is defined as:

Definition 6.2.1.

$$bitsPerAddress = \frac{\left(length_{page} \times size_{page\_element}\right) + size_{directory\_element}}{count_{bits\_set}}$$

For optimal memory usage all elements in the page will be set; for nonoptimal usage only a single element will be set. Figure 6.1 uses an unsigned short for each directory element ($\text{size}_{\text{directory\_element}} = 16$ bits) and a page consisting of a 65536-element bitset ($\text{length}_{\text{page}} = 65536$, $\text{size}_{\text{page\_element}} = 1$ bit) giving an optimal case of 1.0002 bits per address accessed, and a worst case of 8.001 KiB per address accessed. For convenience, page usage efficiency is defined as:

Definition 6.2.2.

$$pageUsageEfficiency = \left(100 \times \frac{count_{bits\_set}}{length_{page}}\right)\%$$

6.2.1 Page Caching

To exploit temporal locality of speculative accesses, this design introduces the use of *page caches*. Page caches are designed to reduce the time taken to find the correct page

for a given speculative access. For each cache the index into the page directory for the most recent speculative access is stored as a tag alongside a pointer to the relevant page. Before accessing the page directory to retrieve the relevant page, a simple check is performed to see if it is the same as the one last accessed using that cache. If so it is accessed directly, bypassing the directory entirely. The assignment of caches to speculative accesses can be performed in various ways, such as assigning a cache to every access of a particular variable or to each speculative access within a loop. Using more caches reduces the likelihood of a cache miss, but increases set-up and memory costs.

6.2.2 Structure Usage

During speculative execution there are three main phases of use of the memory trace structure: allocation/initialisation, trace, and check. The check phase is described in Sections 6.3 to 6.5.

6.2.2.1 Allocation/Initialisation

Initially, the page directory and a suitable number of pages and caches must be allocated in main memory. The page directory must be large enough to cover all possible values within the upper section of address bits of each speculative access. The data type used for each directory entry must be large enough to index through every page in the pool, plus a value to specify that a page has not been assigned. This can be expanded if necessary, at an additional computation cost. The value of each directory entry must be initialised to the value specifying an unassigned page.

The number of pages allocated is code-dependent and difficult to determine at compile time; however the number of pages can be expanded at additional computation cost. If this is likely to occur it may be more appropriate for the page directory to store a pointer rather than index into the pool. Pages must be initialised to an empty bitset.

The number of caches allocated is also code-dependent, but is fixed and can be determined at compile time. Their tag must be initialised with a special value that signifies it does not point to a page. This value cannot match any possible value within the upper section of each speculative access.

```
1  specAccess(Pagetable *pt,uint32 addr,Cache *c) {
2    // Calculate and compare page cache tag
3    uint32 tag = addr & 0xFFFC0000;
4    if (cache->tag != tag) {
5      // If not cached, check if a page has been assigned
6      cache->tag = tag;
7      uint16 *entry = pt->dir + (tag >> 18);
8      if (*entry == pt->unassignedPage) {
9        // If not, assign one, reset it and update cache
10       if (pt->nextPage == pagesAvailable)
11         pt->increasePageTable();
12       *entry = pt->nextPage++;
13       cache->page = pt->pages + *entry;
14       memset(cache->page,0,sizeof(Page))
15     } else
16       // Otherwise just update the cache
17       cache->page = pt->pages + *pageIndex;
18   }
19   // Update the page with the specified access
20   bit = ((addr >> 2) & 0xFFFF);
21   cache->page[bit>>6] |= 1 << (bit & 0x3F)
22 }
```

Figure 6.2: Process of marking a single address as *accessed* in a page table.

To further lower maintenance costs, page tables can be reused for each speculative section of a program. The number of caches allocated can be the maximum required for any given speculative section, and only those required for each section need be reset. Similarly, pages need only be reset when they are first assigned during each speculative section. The page directory *must always* be reset before each speculative section.

6.2.2.2 Trace

During the trace phase of execution, each speculative access is recorded in the page table. The process of recording an access is shown in Figure 6.2. First the tag is

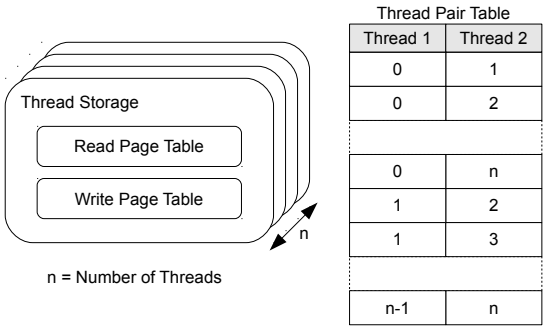


Figure 6.3: Speculative Storage

calculated and compared with the cache for this access (lines 3–4). If the cache does not match then its tag is updated and the entry in the page table is calculated (lines 6–7). The page entry is then examined to determine if a page has been assigned for that location yet (line 8). If not, a new page is assigned and reset (possibly increasing the size of the page table) and the cache is updated with a pointer to that page (lines 11–14). Otherwise, the cache is simply updated with the page already assigned to it (line 17). Finally the bit within the page to be updated is calculated, and then set to mark that it has been accessed (lines 20–21). Should the cache tag match, much of the process is skipped: only the update of the cached page is performed.

6.3 Simple Distributed Error Detection

The simplest error detection scheme is simply to create a memory trace for each thread, and then compare every possible combination of traces to detect a conflict. As shown in Figure 6.3, the topology required for this scheme consists of two page tables per thread: one for reads and one for writes. Each thread's write table must be compared against all other threads' read tables and against all other threads' write tables, thereby detecting all RAW, WAR and WAW dependences. This is performed efficiently by creating a *Thread Pair Table* (TPT), displayed in Figure 6.3, that stores a list of all possible combination of thread pairs. Every pair is distributed as evenly as possible among the available threads. For each pair the first thread's read table is compared against the second thread's write table, then the first thread's write table is compared against the second thread's read table; and finally the write tables of each thread are compared. This method ensures each required comparison occurs only once and that the comparisons are performed *in parallel*. During the check, every page table is a

```
1  collision(Pagetable *pt1, Pagetable *pt2) {
2    uint64 result = 0;
3    uint16 *d1 = pt1->dir, *d2 = pt2->dir;
4    uint64 (*p1)[PAGESIZE] = pt1->pages;
5    uint64 (*p2)[PAGESIZE] = pt2->pages;
6    // Loop over all directory entries
7    for (int i = 0; i < TABLESIZE; i++) {
8      // Inspect to see if both tables have pages assigned
9      if (*d1 != 0xFFFFu && *d2 != 0xFFFFu) {
10       // If so, traverse the pages looking collisions
11       uint64 *x=p1[*d1], *y=p2[*d2];
12       for (j = 0; j < PAGELENGTH; j++) {
13         result |= *x & *y;
14         x++; y++;
15       }
16     }
17     d1++; d2++;
18   }
19   // Return value other than zero indicates a collision
20   return result;
21 }
```

Figure 6.4: Comparing two page tables: if a value other than zero is returned then a collision has occurred.

read-only data structure, thus removing the need for any synchronisation and avoiding cache thrashing during the check.

The comparison of two tables is simple and efficient. Figure 6.4 shows the process for comparing two page tables. Each entry of the page directory is traversed to determine if both tables have a matching entry (lines 7–9). If only one or neither contain entries then a dependence cannot have occurred; otherwise an inspection of both tables pages is required. For error detection purposes it is not a requirement to know which addresses have collided, just whether or not a collision has occurred. This simplifies the inspection of each page into a bitwise `and` of each page. As each page is implemented as an array of unsigned long integers, each element is instead combined using a bitwise `and`, the result of which is bitwise `or`'ed with the earlier comparisons (lines 11–15).

Once the entire directory has been traversed the result is then returned (line 20). Any value other than a result of zero indicates that a dependence has occurred between the two page tables.

6.3.1 Scalability

The scalability of this detection scheme is mostly affected by the increased size of the Thread Pair Table with the addition of more threads. Overall the total number of tables to be compared has, at most, quadratic growth. However, as all pairs are distributed evenly between all threads, the amount of work to be performed by each thread grows linearly.

For example, given four threads, this will result in six pairs, (1-2, 1-3, 1-4, 2-3, 2-4, 3-4) resulting in 1.5 pairs per thread. Adding a fifth thread will result in four extra pairs for a total of ten, or 2 per thread. This can be generalised to $\frac{n(n-1)}{2}$ total pairs, and $\frac{n-1}{2}$ pairs per thread.

6.4 Reduction-Tree Error Detection

A more complex but more scalable method is to perform error detection using a reduction-tree-based algorithm. Instead of examining every possible combination of page tables only adjacent tables are compared. These page tables are then merged together, and each adjacent merged table is compared. This process is repeated until there is only one table left.

Optimal performance of this scheme relies upon an efficient method of merging two page tables. To limit redundant processing the merge of two tables can be performed in-place on two existing tables, with the higher thread always merging into the lower thread. For each element of the page directory this allows for four circumstances:

(i) Neither table's directory entry has a page assigned for that location. No change is required.

(ii) Both tables' directory entry has a page assigned for that location. The page of the higher thread must be merged with the page of the lower thread using a bitwise `or` on each page element.

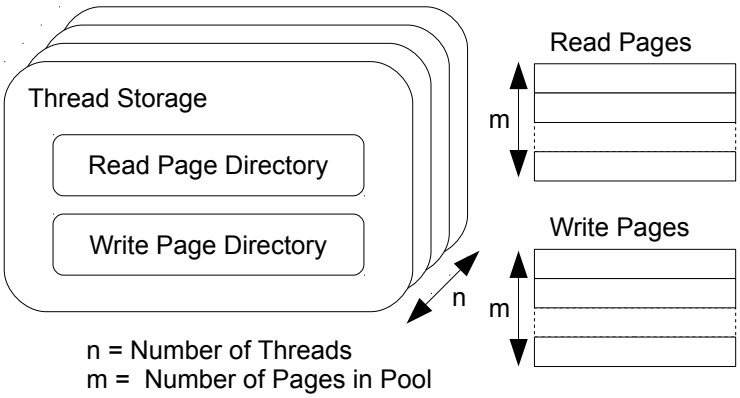


Figure 6.5: Tree Storage Topology

(iii) The table with the lower ID has a page assigned to that directory entry, but the thread with the higher ID does not. No change is required.

(iv) The table with the higher ID has a page assigned to that directory entry, but the thread with the lower ID does not. The page assigned to the higher thread should be reassigned to the lower thread.

To assign a page from one table to another requires a small modification to the existing page-table data structure. Specifically, the pool of available pages for each table must be merged into a central pool of pages of equal size, allowing for each directory to refer to any page assigned by any thread during execution. No additional synchronisation is required as each thread can be assigned its own section of pages within the pool for use during execution, with indexing to pages outside of that region only occurring during merges. The storage topology implementing this change is shown in Figure 6.5.

To benefit from the caching methods used in modern CPUs the merge should be performed on-the-fly with the check. This ensures any directory elements or page elements that must be merged have already been loaded into the CPU cache when it is to be modified. The alternative would involve loading it to perform the check, having it be evicted from the cache by the rest of the check, and then reloading it to perform the merge.

The check is performed in two stages. First a list of page pairs is generated for entries that require checking, merging, or both. In the second stage the list of pairs is processed, checking and merging as appropriate. This two-stage process is used to distribute all work evenly between the threads, ensuring the most efficient processing

of each pass.

During the first stage each thread is assigned a chunk of directories to process. Chunks are distributed evenly among all threads. Figure 6.6 demonstrates how the scan of a chunk of directory entries is performed. The thread processes each directory entry in its chunk individually (line 6). First it inspects the read entries and write entries from each table extracting the a pointer to each page and filling a state variable (lines 9–16). The state variable is used to determine what further action, if any, is required. For instance, if both directory entries have write pages assigned, then there could be a write dependence logged in the two pages. In that case (state == 3) the two pages would need to be scanned for dependences and merged together for the next pass. After the pointers and state variable has been calculated, the state is checked to see if further processing is required (lines 19–22). If so the state and page indexes are added to the list (lines 23–24). In some cases the pages may need to be reassigned instead of merged which is performed during the first stage (lines 23–31). At the end the count of how many page pairs require extra processing is returned (line 36).

During the second stage each thread processes a section of each pair of pages distributed evenly among all threads. Figure 6.7 demonstrates how a thread processes a pair of pages. Each thread is passed in the list of pages to be processed (`list`), the number of items in the list (`count`), the start index within the page that the thread should check from (`start`) and the quantity of page elements that it must process (`size`). The thread handles each list item individually (line 4). First a pointer to the first element the thread must process is extracted for each page (lines 6–7). Next the state variable is inspected to select the most efficient loop for scanning/merging the set of pages (line 8). There are nine different combinations of read and write pages where different operations must be performed on the pages themselves. Examining the case where all directory elements have pages assigned (Case 15), first the read and write pages are compared using a bitwise `and` on each element `or`ed together (line 21). The necessary pages are then merged together into the table with the lower ID, again using a bitwise `or` (line 22). This action is performed across the entire section of the page assigned to the thread. Finally the result is returned with any non-zero value indicating an error.

```
1  fillCollisionTable(uint16 *rdD1, uint16 *rdD2,
2                     uint16 *wrD1, uint16 *wrD2,
3                     struct PageCollision *list, int size) {
4    int count = 0;
5    // Traverse all directory entries assigned
6    for (int i = 0; i < size; i++) {
7      struct PageCollision item; item.state = 0;
8      // Get allocated pages and update state variable
9      if (*rdDir1 != 0xFFFFu) { item.r1=pages[*rdD1];
10                               item.state += 8; }
11     if (*rdDir2 != 0xFFFFu) { item.r2=pages[*rdD2];
12                               item.state += 4; }
13     if (*wrDir1 != 0xFFFFu) { item.w1=pages[*wrD1];
14                               item.state += 2; }
15     if (*wrDir2 != 0xFFFFu) { item.w2=pages[*wrD2];
16                               item.state += 1; }
17     // If both tables could have a dependence violation
18     // Then add their pages to the list to be scanned/merged
19     if ((state|12) | (state|3) | (state|9) | (state|6) != 0) {
20       *list++ = item;
21       count++;
22     }
23     // If higher thread has a page but lower doesn't
24     // Then reassign higher page to lower for both read and write
25     if (item.state | 3 == 1) // __01
26       *wrDir1 = *wrDir2;
27     if (item.state | 12 == 4) // 01__
28       *rdDir1 = *rdDir2;
29     // Move onto the next directory entry
30     rdDir1++; rdDir2++; wrDir1++; wrDir2++;
31   }
32   // Return number of potentially colliding pages found
33   return count;
34 }
```

Figure 6.6: Stage one of the reduction-tree process: two page tables are compared. Any potentially colliding entries, for instance when both tables have write pages assigned, are added to a list for further processing.

```
1  processList(struct PageCollision *list, int count,
2              int start, int size) {
3    uint64 result = 0;
4    // Traverse each set of pages in the collision list
5    for (int i = 0; i < count; i++) {
6      // Get pointers for where to start comparing/merging
7      uint64 *r1 = list->r1 + start, *r2 = list->r2 + start,
8             *w1 = list->w1 + start, *w2 = list->w2 + start;
9      switch(list->state) {
10      case 3: // 0011
11       // Traverse each page element
12       for (int j = 0; j < size; j++) {
13         // Inspect for collisions
14         out |= (*w1 & *w2);
15         // Merge element contents
16         *w1 |= *w2;
17         w1++; w2++;
18       } break;
19      // Cases 0110, 0111, 1001, 1011, 1100, 1101, 1110 omitted
20      case 15: // 1111
21       for (int j = 0; j < size; j++) {
22         out |= (*r1 & *w2) | (*r2 & *w1) | (*w1 & *w2);
23         *r1 |= *r2; *w1 |= *w2;
24         r1++; r2++; w1++; w2++;
25       } break;
26     }
27     // Move onto the next element in the list
28     list++;
29   }
30   // A return value other than zero indicates a collision
31   return result;
32 }
```

Figure 6.7: Stage two of the Reduction-Tree Process: each potentially conflicting pair of pages are scanned and merged as appropriate. If a value other than zero is returned then a collision has occurred.

6.4.1 Scalability

The main influence on scalability with this model is how much extra processing is required on the addition of more threads. As this is a tree-based detection algorithm this is centred around how many compare and merge passes are performed. Definition 6.4.1 states how many passes are required to perform a full dependence check.

Definition 6.4.1.

$$totalPasses = \lceil \log_2 (num\_threads) \rceil$$

As can be seen the number of passes required grows logarithmically with the addition of more threads. Additionally the most optimal detection scheme is one with a thread count that is a factor of two, so that each stage of the reduction tree is fully utilised.

6.5 GPU Conflict Detection

By removing the ability to checkpoint and rollback from a TLS system it is possible to offload dependence checking to an auxiliary device. Such offloading allows the main program to continue to execute during the dependence check, minimising the delay caused by performing the check. This section presents a conflict-detection scheme using a GPGPU as an auxiliary processing unit.

The execution workflow for TLS on a GPU differs from that of a CPU. Whilst executing speculative regions of code, memory access traces are recorded normally as with CPU-based schemes. At the end of the speculative region, the memory logs must be transferred to the GPU or placed in a region of memory accessible by the GPU. The check is then initiated on the GPU and, once completed, the result must be copied back to CPU-accessible memory and used to determine whether the process must be cancelled or may be allowed to continue. Modern GPGPU frameworks such as CUDA or OpenCL allow all interactions with a GPU to be placed in a queue to be executed automatically when the previous operations have finished, allowing the main thread to continue to execute once all stages have been added to the queue.

Writing and optimising GPGPU-based programs can be notoriously difficult, requiring very specific design considerations. For convenience these can be split into two categories: CPU- or host-side optimisations and GPU- or device-side optimisations.

For the host-side, considerations include:

(i) Where possible, delays and barriers on the critical path must be minimised. At the end of a speculative section all actions to be performed on the device must be added to the device queue (or passed to another thread to do so) before the main thread can continue.

(ii) It is desirable for additional speculative regions to be executable while the previous speculative section is still being checked. This can be done using a pool of speculative storage structures used in a round-robin fashion. While one is being copied to the device, another can be filled by a speculative region. Each structure can also be copied to device memory while the previous check is being executed on the device, effectively hiding the costs of all data transfers. Finally, if the device is at capacity, it is possible to fall back to one of the host-based checks from the previous sections.

(iii) Memory transfers from host to device are expensive, so it is desirable to minimise the amount of data transferred to the device to the extent possible. Instead of transferring the entire pool of pages to the device, it is possible to only transfer those that were used during the speculative execution.

(iv) Interactions with the GPU, such as adding tasks to the queue or using callbacks, are similarly expensive. Where possible, memory transfers for each thread should be performed in bulk, resulting in only a single task added to the queue instead of one per thread. To enable this when transferring only used pages, each thread must access a centralised pool with a mutex locked counter for the next page.

For the device-side, considerations include:

(i) Each device has a maximum number of threads and thread blocks that can be executed at once. The check must be configured such that all threads are used at once without going over the thread block limit.

(ii) Thread instructions on the device are interlocked within each thread block and hence execute every diverged path, even if only one thread within the block uses that path, so divergence must be minimised. Traversing a page table by the directory is inherently diverse as each entry may or may not contain a corresponding page. Traversing by each page results in wasted computation where each thread repeatedly examines the directory for that page. This is a tradeoff that must be considered during optimisation.

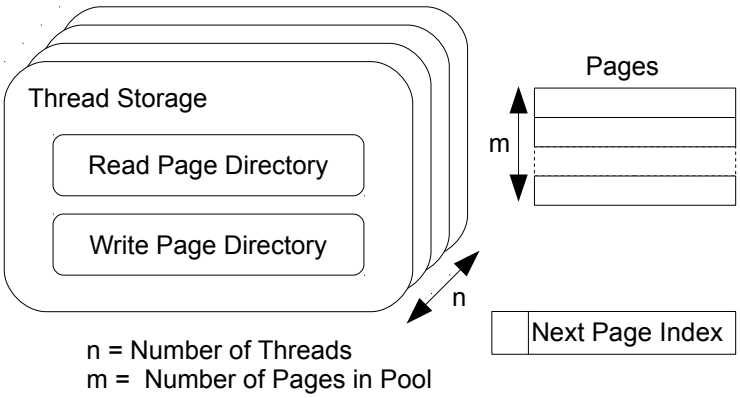


Figure 6.8: GPU Storage Topology

(iii) Thread-local memory is much faster than global memory but more limited in size. Where possible, especially for interactions between threads, thread-local memory should be utilised.

(iv) Accesses to global memory should be coalesced where possible. Due to the design of device memory, each memory access to an individual address triggers access of a chunk of addresses. If each thread performs an access in a coalesced fashion then every access can be performed in the same chunk.

The resulting data structure used can be seen in Figure 6.8. Similar to the reduction-tree checking scheme, each thread maintains its own page directory and each directory indexes into a centralised pool of pages. However, now the index for the next available page is maintained through a centralised, mutex-locked counter on the host. This adds synchronisation costs when a page is allocated, while dramatically reducing the amount of data to be transferred to the device. Additionally there are now multiple data structures the host can access to allow the pipelining of checks on the device.

The check performed on the device is similar to that of the reduction-tree-based check from the previous section. First a list of pages requiring further action is generated; then each set of pages in the list is processed. In this design, thread blocks are determined by the directory being compared, with all threads in a block processing a section of the same page directory during the first section, and all threads in a block processing the same page during the second section. Multiple thread blocks are able to execute at once allowing for multiple directory pairs to be processed simultaneously. Similar to the CPU-based reduction tree, each full check requires multiple stages of

checking and merging, corresponding to each level of the reduction tree. All stages of the reduction tree are queued at the start, but subsequent stages cannot be started until all thread blocks from a previous stage have completed executing.

Figure 6.9 shows the process for completing the first section on the device. First the IDs are calculated to determine which two directories are to be compared and merged (lines 6–7). This is based on which stage of the reduction is being processed, with the first stage pairing directories of threads 0 and 1, 2 and 3, and so on, and the second stage pairing directories 0 and 2, 4 and 8 and so on until every stage is completed. Next the list of pages requiring further processing is created and initialised (lines 9–10). This list is stored in local memory and shared between all threads in the thread block. A barrier is required at this point to ensure that the list has been fully initialised before other threads begin to access it.

After this point the directories are processed on the device. Each thread within the block handles a single row in each directory; these are read into thread-private memory by lines 15–16. Once read, a state variable is calculated to determine the required course of action for this page. This is stored in short bitset to allow for easier comparison at later stages (lines 18–19). Next it is determined whether this combination of directories requires additional processing in the form of checks and/or merges to be performed in the second section. If so, a free index into the list is safely acquired through an atomic increment function, and the element at that list is updated (lines 21–25). The atomic operation ensures safe shared use of the list and is efficiently implemented in device hardware; however it does add a small cost to performance. The final part of the first section is to copy over any relevant page indexes when a full merge is not required (lines 27–28). After that, section two of operation commences (line 31) and all pages that were added to the list are processed, as described below. This entire process is repeated until all items in each directory have been processed. At that point the output variable is updated if necessary, and execution finishes.

Figure 6.10 shows the process for completing the second section. For this section all groups of pages that require additional actions are processed accordingly. Each item in the list is read in from local memory along with the state variable calculated earlier to determine what action is required (lines 10–11). Once the correct operations have been determined (line 13), each page is then split and distributed between all threads in the block to ensure an evenly distributed workload between the threads. For instance, state 3, or two write pages would require both pages to be compared against each other

```
__kernel void checkKernel(__global uint16 (*dirs)[2][DIRLEN],
                          __global uint64 (*ps)[PAGELEN],
                          __global bool * output, int stage) {
  uint16 rd1, wd1, rd2, wd2, s; uint64 out = 0;
      // Calculate which directories to check
  int d1 = get_global_id(0) * (1 << (stage + 1));
  int d2 = dir1 + (1 << (stage + 1)) / 2;
      // Create Local List of Page Groups
  __local int listCounter; __local uint16 list[BLOCKSIZE][5];
  if (get_local_id(1) == 0) listCounter = 0; barrier(LOCAL_MEM);
      // Check each directory entry
  for (int i = get_local_id(1); i < DIRLEN; i += BLOCKSIZE) {
    s = 0;
      // Get directory entries
    rd1 = dirs[d1][0][i]; wd1 = dirs[d1][1][i];
    rd2 = dirs[d2][0][i]; wd2 = dirs[d2][1][i];
      // Calculate state var
    if (rD1 != 0xFFFFu) s += 8; if (rd2 != 0xFFFFu) s += 4;
    if (wD1 != 0xFFFFu) s += 2; if (wd2 != 0xFFFFu) s += 1;
        // If pair needs check/merge add to list
    if ((s | 12) | (s | 3) | (s | 9) | (s | 6) != 0) {
      int j = atomic_inc(&listCounter);
      list[j][0] = rd1; list[j][1] = rd2; list[j][2] = rd1;
      list[j][3] = rd2; list[j][4] = s;
    }
      // Reassign page indexes to lower thread
    if (s | 3 == 1) dirs[d1][1][i] = wd2; // __01
    if (s | 12 == 4) dirs[d1][0][i] = rd2; // 01__
    barrier(LOCAL_MEM);
      // Process any items in the list
    out |= handleList(ps, list, listCounter);
      // Reset List
    if (get_local_id(1) == 0) listCounter = 0; barrier(LOCAL_MEM);
  }
  // If collision detected, update output
  if (out != 0) *output = true;
}
```

Figure 6.9: Comparing two page tables using the reduce-merge scheme: if a value other than zero is returned then a collision has occurred.

```
1  uint64 handleList(uint64 (*pages)[PAGELEN],
2                    __local uint16 (*list)[5],
3                    int listCounter) {
4    uint16 rdDir1, rdDir2, wrDir1, wrDir2, state;
5    uint64 output = 0, r1, r2, w1, w2;
6    int i, j;
7      // Traverse each page group
8    for (i = 0; i < listCounter; i++) {
9        // Get page indexes and state
10     rdDir1 = list[i][0]; rdDir2 = list[i][1]; wrDir1 = list[i][2];
11     wrDir2 = list[i][3]; state = list[i][4];
12       // Select most efficient check/merge loop
13     switch (state) {
14      case 3: // 0011
15         // Traverse each page element
16       for (j = get_local_id(1); j < PAGELENGTH; j += BLOCKSIZE) {
17           // Get actual page elements
18         w1 = pages[wrDir1][j]; w2 = pages[wrDir2][j];
19           // Inspect for collisions
20         output |= w1 & w2;
21           // Merge element contents
22         pages[wrDir1][j] = w1 | w2;
23       } break;
24      // Cases 0110, 0111, 1001, 1011, 1100, 1101 & 1110 omitted
25      case 15: // 1111
26       for (j = get_local_id(1); j < PAGELENGTH; j += BLOCKSIZE) {
27         r1 = pages[rdDir1][j]; r2 = pages[rdDir2][j];
28         w1 = pages[wrDir1][j]; w2 = pages[wrDir2][j];
29         out |= (r1 & w2) | (r2 & w1) | (w1 & w2);
30         pages[rdDir1][j] = r1 | r2; pages[wrDir1][j] = w1 | w2;
31       } break;
32     }
33   }
34     // Returning any value other than zero indicates a collision
35   return output;
36 }
```

Figure 6.10: Comparing two page tables using the reduce-merge scheme: if a value other than zero is returned then a collision has occurred.

for access collisions (line 20), and then merged together ready to be processed during the next stage (line 22). The check and merge process is very similar to that of the CPU-based reduction tree mentioned previously.

The two main considerations of this design are thread divergence and memory access coalescing. During each section every access to global memory is coalesced together with the one exception of lines 27–28 in Figure 6.9. This ensures that all accesses to global memory are performed as efficiently as possible by the device. The comparison of page directories contains an inherent thread divergence, due to the comparison of whether or not pages have been assigned. Through this design all thread divergence is restricted to the execution of the first section (lines 18–28), which has been optimised to contain only minimal processing, thus minimising the cost caused by thread divergence. Once the list has been populated all processing required is distributed evenly among threads and ensures no divergence amongst threads within each block.

6.6 Automated Program Transformations

For any auto-parallelisation system, analyses and transformations must be applied to existing code to discover and exploit parallelism that may exist. Those systems using purely static analysis to discover parallelism often require only the addition of threading code, synchronisation points and privatised variables when required. In contrast a particularly aggressive paralleliser using profiling to discover parallelisation might create unsafe code if it were to only perform the same transformations as a purely static auto-paralleliser. It is at these *aggressive* parallelisers that the design presented in this chapter is aimed. Parallelisable loops were selected using the profiling techniques presented in [44]. Discovered probably-parallel loops were marked up using the OPENMP parallel framework, with privatised and reduction variables added as appropriate. This section discusses the analyses and transformations used to implement error-checked execution on these marked up loops.

After the discovery of *probably-parallel* loops, code transformations are then applied to the original sequential application. The code required to perform speculative execution has been reduced to a library of functions, which our new application links to. This keeps code transformation simple and automatable. An example transformation of a simple sequential loop into a speculatively-parallel loop is shown in Figure 6.11.

```
for (int i = 0; i < N; i++)
{
  a[b[i]] += a[c[i]];
}
```

(a) Original Sequential Loop

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
  a[b[i]] += a[c[i]];
}
```

(b) Unsafely Parallelised Loop

```
#pragma omp parallel
{
  __EC_READ_INIT(c);
  __EC_READ_INIT(b);
  __EC_READ_INIT(a);
  __EC_WRITE_INIT(a);
#pragma omp for
  for (int i = 0; i < N; i++)
  {
    __EC_READ(c[i], c);
    __EC_READ(b[i], b);
    __EC_READ(a[b[i]], a);
    __EC_WRITE(a[c[i]], a);
    a[b[i]] += a[c[i]];
  }
  __EC_CHECK();
}
```

(c) Error-Checked Parallel Loop

```
#pragma omp parallel
{
  __EC_READ_INIT(a);
  __EC_WRITE_INIT(a);
#pragma omp for
  for (int i = 0; i < N; i++)
  {
    __EC_READ(a[b[i]], a);
    __EC_WRITE(a[c[i]], a);
    a[b[i]] += a[c[i]];
  }
  __EC_CHECK();
}
```

(d) Optimised Error-Checked Parallel Loop

Figure 6.11: Example code transformations required to turn an unsafely parallelised loop into an error-checked parallel loop. `a`, `b` and `c` are all unaliased arrays containing at least `N` elements.

Each probably-parallel loop is first enclosed in an OPENMP `parallel` section which indicates that the enclosed section should be executed in parallel. Next, each variable that is being speculatively tracked is specified explicitly; this is done using `__EC_READ_INIT` and `__EC_WRITE_INIT` statements which assigns caches to each variable. The loop itself is then surrounded by an OPENMP `for` directive which indicates that the iterations of the following `for` loop should be distributed among the threads. The loop is then traversed and every access of a given variable is preceded by a `__EC_READ` or a `__EC_WRITE` statement to mark the address as being accessed. Finally, a call to the speculative check function, `__EC_CHECK`, is inserted after the loop's execution, still within the `parallel` region.

Tracking the memory accesses performed is an expensive process, so it is desirable that we only track *dangerous* accesses — those that can potentially cause a dependence violation. To do so, limited static analysis can be performed on the program to generate a list of variables that are considered dangerous. It is a waste of computation to track the accesses of:

Read-only variables. If a variable only ever has its value read and not written to then it is impossible for that variable to directly cause a data dependence. Therefore tracking code is not added to completely read-only variables. However, when pointers or references are used it is possible that what appears to be a read-only variable may be *aliased* to another variable that is not, potentially causing a dependence violation.

Thread-private variables. Access to variables that have been privatised and are not subsequently shared between threads cannot directly cause a data dependence. However, if these variables contain pointers to other data then it is possible that access of the data they point to could cause a data dependence.

To scan for truly read-only variables, an Abstract Syntax Tree (AST) of the entire program is generated and traversed looking for pointer and reference interactions. If two variables could share pointers then those variables are linked to each other and assumed that they could be the same variable. These links are shared between other linked variables creating a graph of variables that could potentially be aliased. This is a conservative approach assuming that if they might alias each other then they will. For instance, if variable `a` shares a pointer with variable `b`, and then `b` shares a pointer with variable `c` then variables `a` and `c` are also linked together. Once the graph of

linked variables has been generated, it is then possible to determine which variables are definitely read-only during a probably-parallel loop. Note that this conservative approach to static alias detection on read-only variables does not degrade accuracy of dependence detection. As static analysis is only used as a filter to reduce the overhead of dependence profiling, any overly eager classification of an alias will be still captured during the profiling stage. Thread-private pointers are handled in the same manner.

6.7 Empirical Evaluation

Due to restrictions on the available hardware at the time of evaluation, each scheme has been split into two groups:

1. CPU-only schemes
2. GPU schemes

Each group targets a standard shared memory system. In the case of CPU-only schemes this consists of four 8-core Intel XEON processors, and for the GPU scheme, two 6-core Intel XEON processors with an NVIDIA Tesla K20 GPU. A full overview of each target platform is available in Table 6.1.

For the evaluation of each speculation scheme, benchmarks have been selected from the NAS Parallel Benchmark suite [2]. For these benchmarks there exist both standard sequential and manually parallelised implementations. Each benchmark was speculatively parallelised using a profiled execution based scheme [44]. Loops detected as probably parallel were then selected if their execution time consisted of greater than 1% of the total sequential execution time. Additionally, loops that could be proven parallel using an existing static analysis tool [13] were parallelised without speculative error-checking. An overview of each benchmark is available in Table 6.2.

Each benchmark was executed in a number of different configurations to determine how well each speculative scheme performed:

Sequential

This is the baseline measurement that each speedup was compared to.

Manual Parallel

This is the upper bound of the performance obtainable by each benchmark when

	CPU-only Schemes	GPU Scheme
Host	Quad-Socket, Intel Xeon L7555 @ 1.9GHz 4 × 8-cores, 32 cores total 3MB L3-cache shared/8 cores (24MB/chip) 64GB DDR3 SDRAM @ 1333MHz	Dual-Socket, Intel Xeon E5-2620 @ 2.0GHz 2 × 6-cores w/ Hyperthreading 12 cores total, 24 apparent threads 2.5MB L3-cache shared/6 cores (15MB/chip) 16GB DDR3 SDRAM @ 1333MHz
Device	N/A	NVIDIA Tesla K20 2496 cores @ 706 MHz 5GB GDDR5 SDRAM @ 2600MHz
OS	openSUSE 12.3 (x86_64) kernel 3.7.10-1.1	
Compiler	gcc 4.7.2 build 20130108 -O3 -fopenmp -lpthread -lm	-O3 -fopenmp -lOpenCL -lpthread -lm OpenCL Version 1.1

Table 6.1: Hardware and software configuration details of the target platforms.

Benchmark	Static Loops	Probably-Parallel	Selected Loops	Sequential Time
Block Tridiagonal (BT)	30	20	8	242.31
Conjugate Gradient (CG)	11	3	2	2.23
Embarassingly Parallel (EP)	0	1	1	53.38
Fast Fourier Transform (FT)	2	3	3	7.75
Integer Sort (IS)	0	1	1	1.08
Lower-Upper Symmetric Gauss-Seidel (LU)	19	6	6	89.84
MultiGrid (MG)	0	11	6	3.26
Scalar Pentadiagonal (SP)	0	60	21	102.70

Table 6.2: Benchmark statistics including (a) number of statically detected loops, (b) number of additional probably-parallel loops detected, (c) number of probably-parallel loops using $>$1% of the sequential execution time, (d) the sequential execution time in seconds when run on the CPU-only platform.

hand tuned.

Unsafe Parallel

This is the automatically parallelised version using the profiling technique mentioned above. This execution is unsafe as static analysis has not proven that data dependences do not exist; however, it is performed to measure the possible speedup obtainable without speculative support.

Trace Only

This is the automatically-parallelised version, with memory traces recorded but not scanned for data dependences. This is to provide analysis of the impact on performance that the logging mechanism has.

Full Error Detection

This is the full error detection scheme at work.

Statistics Collection

This is a non–performance-based execution to collect statistics on how each model performs.

Initially the performance of the parallelisation technique is evaluated in Section 6.7.1. This is followed by an analysis of page table statistics and the impact of recording the memory trace in Section 6.7.2. Each CPU-only scheme is then evaluated and compared in Sections 6.7.3 and 6.7.4, with an evaluation of the hybrid CPU-GPU scheme in Section 6.7.5. Finally, the impact of and solutions to incorrectly parallelising a loop containing data dependences are discussed in Section 6.7.6.

6.7.1 Auto-Parallelisation Analysis

The speedups obtained using the automated parallelisation technique as executed on the CPU can be found in Figure 6.12. As can be seen, an appreciable speedup is obtained for all but one benchmark with speedups of up to 23.8 times with a geometric mean of 4.42 times that of sequential execution when executing with 32 threads. In four of the benchmarks (BT, CG, EP and FT), it is notable that the performance continues to increase with the addition of further threads. However, for three of the benchmarks (IS, MG and SP) the performance increase peaks at around 8–16 threads.

Figure 6.13 displays the speedup of the automated parallelisation technique compared

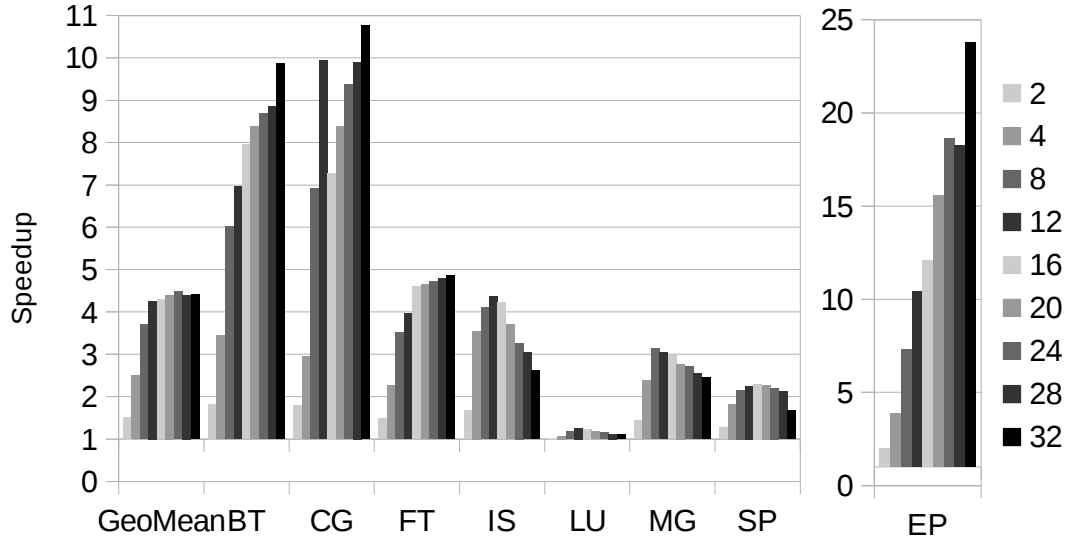


Figure 6.12: Performance of each automatically-parallelised benchmark executed on the CPU compared to sequential execution.

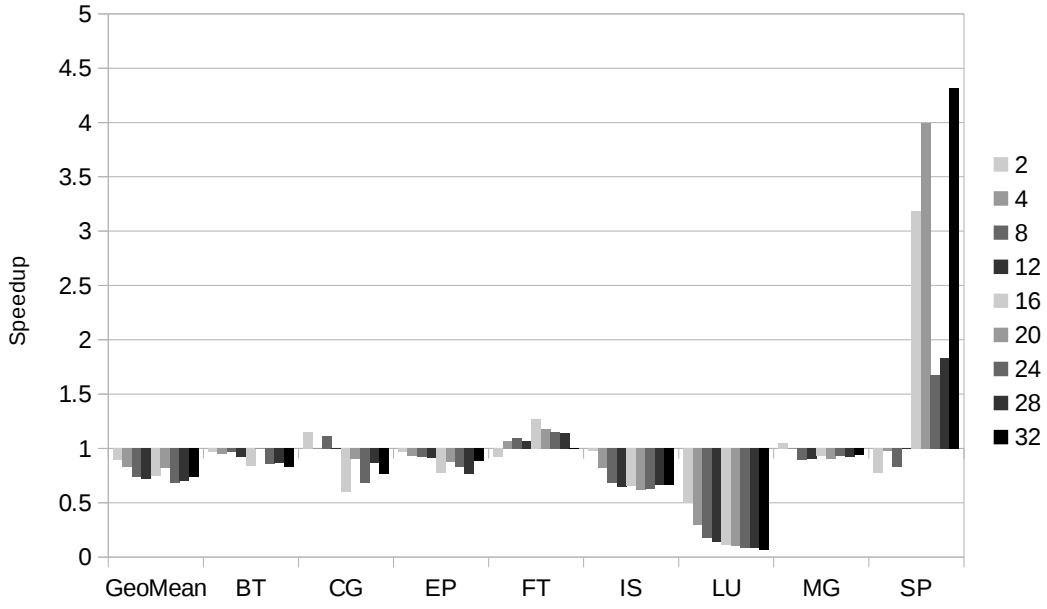


Figure 6.13: Performance of each automatically-parallelised benchmark compared to manually parallelised execution.

Benchmark	Loops	Pages Used	Memory Used	Access Efficiency
BT	8	11,744	92.75 MiB	99.07%
CG	2	32	1.25 MiB	99.77%
EP	1	64	1.5 MiB	100.00%
FT	3	16,384	129 MiB	97.52%
IS	1	928	8.25 MiB	61.65%
LU	6	1,440	12.25 MiB	98.20%
MG	6	1,856	15.5 MiB	58.54%
SP	21	1,664	14 MiB	99.03%

Table 6.3: Page usage statistics for 32-thread execution.

to the manually parallelised version. In most executions the automatically parallelised version has worse performance than the manual version. This is unsurprising, as many of the manual versions involve more complex or coarser parallelisation techniques and have been fine tuned for optimal performance. This performance is nonetheless in the range of 0.8–0.9 times the speed of the manual version, which still constitutes an appreciable speedup over serial execution. It is also notable that in some cases the automated process provides performance increases over that of the manual version. An exceptional instance is the SP benchmark, which provides a speedup of up to 4.31 times over manual parallelisation. This appears to be due to the coarser-grained parallelism that the manual version applies, introducing slowdowns associated with certain parallelised sections, whereas the automated version simply executes those sections sequentially. In contrast the LU benchmark has the opposite issue, in that the fine-grained parallelism implemented by the automated version introduces too many overheads for each parallel section, thus preventing substantial speedups from being achieved.

6.7.2 Page Table Statistics

The efficacy of the page table structure can be evaluated in a number of ways. The most important issues are: the impact it has on the overall memory footprint of each benchmark; how efficiently each access is recorded; and how much time is spent interacting with the page table versus actual execution of the benchmark.

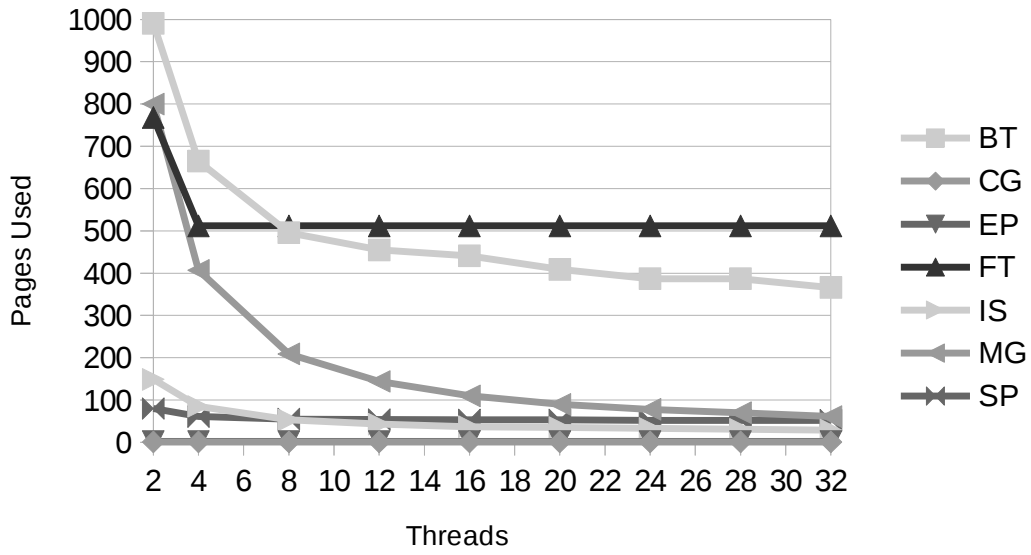


Figure 6.14: Pages used per-thread based on thread count

Due to the dynamic nature of the page table structure the impact it has on memory usage varies depending on the program that is using it. A longer running parallel section will likely interact with larger sections of memory, generating a much larger trace than that of smaller parallel sections of a program. Table 6.3 presents the total number of pages used and the overall memory usage caused by each benchmark when running at 32 threads. As can be seen there is a wide variation in the number of pages used by each benchmark ranging from as low as 32 pages (1 per thread) up to 16,384 (512 per thread). This obviously has a direct effect on the amount of memory required to perform each trace again, ranging from as low as 1.25 MiB up to 129 MiB taking into account both pages and directories. On average the memory footprint was 34.3 MiB.

The configuration of the page table used during evaluation was word-based tracking within a 32-bit address space. This means that the page table was able to distinctly identify between 2^{30} different addresses. To provide perspective on the memory footprints mentioned above, a standard bitset would require 128 MiB of space per thread, or 4 GiB of storage space for 32-thread execution, on average 119 times the required space compared to the page table structure. Comparing this to a more complex, centralised tracking structure such as that used by SpLSC, for both read and write vectors using a short int as an access counter would again require 4 GiB of stage space. Programs using SpLSC would likely use techniques discussed in Section 2.3.6 to reduce

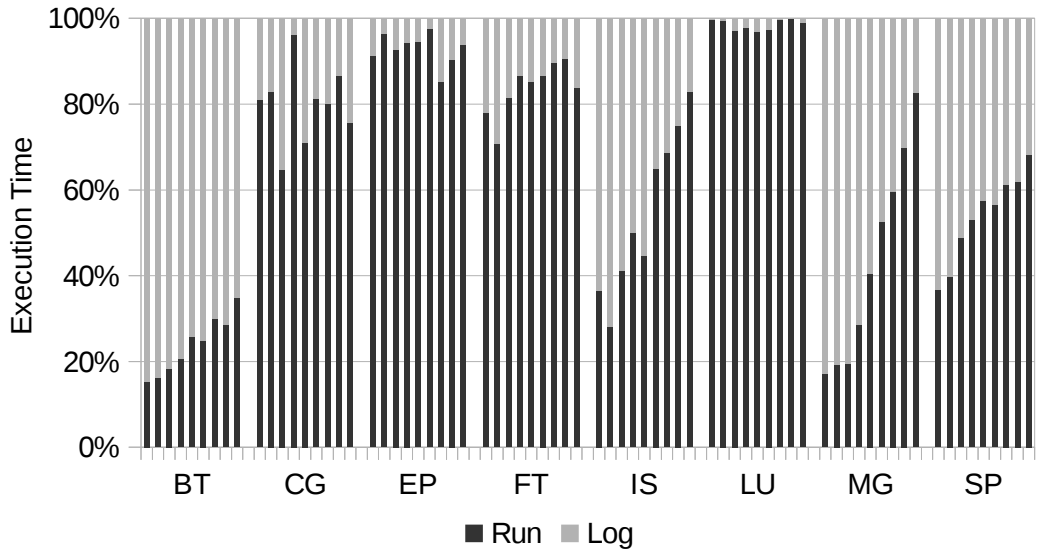


Figure 6.15: Execution breakdown during memory traces. Each bar represents a separate thread count at 2, 4, 8, 12,16, 20, 24, 28 and 32 left-to-right for each respective benchmark.

this size dramatically; however, doing so without introducing the risk of false data dependences requires advanced knowledge about the memory access patterns of each program, adding significant complexity to the parallelisation process. This process has also not yet been automated. In contrast, the page-table design automatically scales as required and cannot trigger false dependences.

It is possible to look more closely at how each benchmark interacts with the page table by analysing the page usage across a different range of executing threads. Figure 6.14 presents the maximum number of pages used by an unsafe parallel section for each benchmark. In general the lower the quantity of threads executing, the larger number of pages each thread interacts with. This indicates that as the number of threads grows, the amount of additional memory required for memory traces shrinks. Notable exceptions to this are when the threads already require a low number of pages, CG and EP, and for the benchmark FT which remains constant per thread beyond four threads.

Page caches were introduced as a means to increase the efficiency of every access by bypassing the lookup of the page directory. This was implemented by maintaining a list of {`tag, pointer`} tuples, in this evaluation at a per-variable level. To inspect how well page caches performed first requires a definition for an inefficient access. An

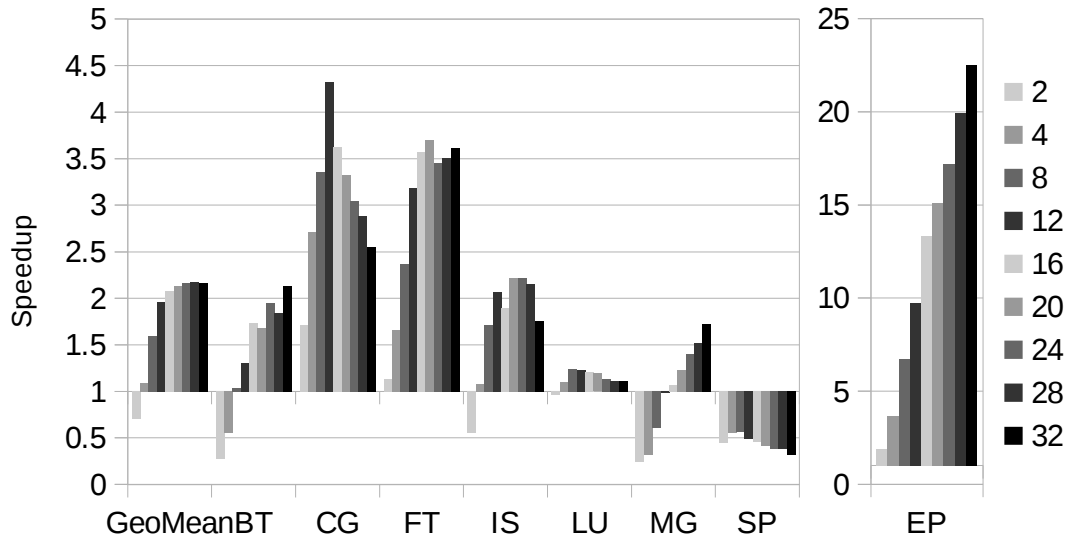


Figure 6.16: Speedup of simple distributed detection scheme

inefficient access is one that causes a speculative cache miss; that is the most recent speculative access for that variable occurred in a page different from the most recent access. From this, cache efficiency percentage e_{cache} can be defined as:

$$e_{\text{cache}} = 100 - \frac{100 \times \#\text{ inefficient accesses}}{\#\text{ total accesses}}$$

Table 6.3 provides the overall cache efficiency for each benchmark at 32-thread execution. As can be seen overall access efficiency is high with six of the eight benchmarks performing at over 97% efficiency, and on average 89.22%. It is likely that for the less accurate benchmarks, the memory traces could be used to identify which accesses were causing cache misses and assign them their own cache, thereby further increasing efficiency.

The final part of the page table evaluation relies upon how much time is spent recording accesses compared to the amount of time spent executing the actual benchmarks. Figure 6.15 presents this ratio for each benchmark executing at a number of different thread levels. As can be seen the ratio is highly dependent upon the benchmark itself, with some benchmarks performing many more speculative accesses during each parallel section than others. For those benchmarks that do not perform a large number of speculative accesses (approximately half of them) the time spent recording each access

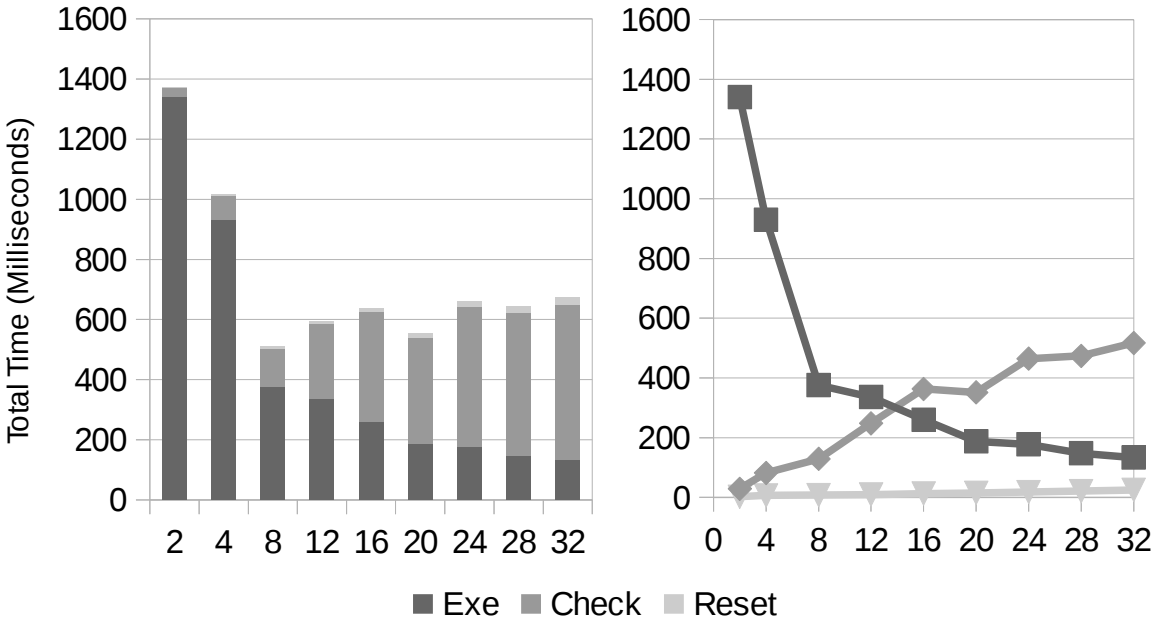


Figure 6.17: Execution time analysis for *CG* benchmark.

is relatively low — between 10 and 30% of the overall execution. For the other half the impact of recording the memory traces can be quite high, with up to 85% of the execution time being spent performing these traces. However in each case it is clear that as the number of threads grows the impact of performing the trace decreases, with the majority of the benchmarks performing within the original 10–30% range.

6.7.3 Simple Distributed Detection Scheme

Figure 6.16 provides the overall speedups of each benchmark across a range of thread counts using the simple distributed detection scheme. The scheme managed to achieve a speedup of up to 22.53×, with a geometric mean of 2.16× that over sequential execution with 32 threads (1.55× excluding EP). It is noticeable that speculative execution scales better with certain benchmarks over others. Notably BT, EP and MG all continue to profit from introducing additional threads. CG, FT, IS and LU all appear to hit a peak performance at 12–20 threads, while for CG and FT their counterpart results when run unsafely continue to improve with more threads. This suggests that the checking mechanism is becoming a dominant part of the execution time. If EP is

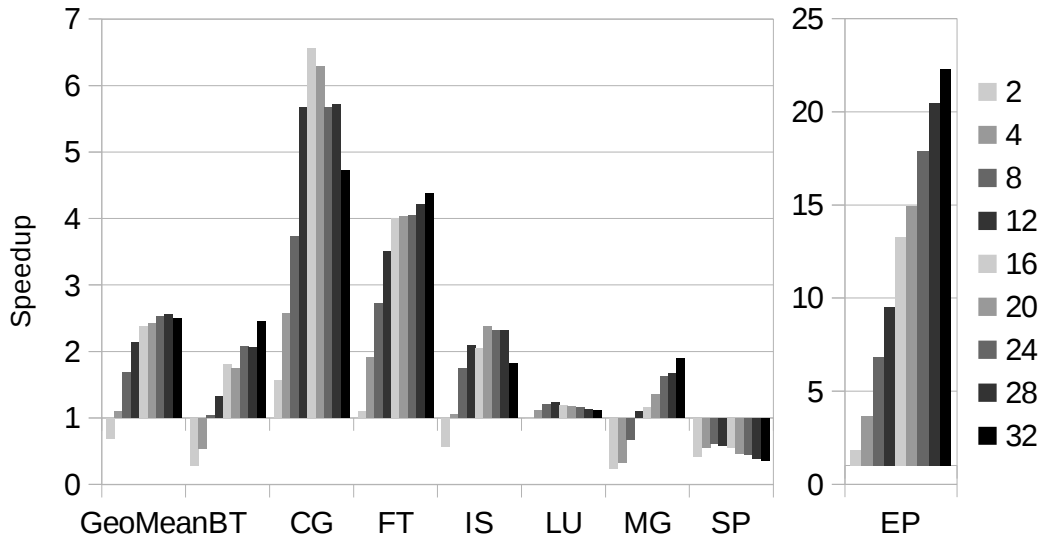


Figure 6.18: Speedup of reduction-tree detection scheme

excluded, the average speedup peaks at 1.61× at 20 threads.

Only one benchmark consistently performs poorly, SP. Further analysis indicates that this is due to the memory access patterns of each thread. During loop execution practically all accesses are performed in the same memory region by each thread, so they all have pages assigned to the same regions. This causes an expensive check whereby every thread has conflicting pages that must be scanned slowly. This benchmark would benefit from smaller pages to allow for fewer conflicts, or from a centralised trace structure allowing for a more optimised *on-the-fly* check.

Investigating the execution further, it is noticeable that the time spent checking for dependence violations becomes the limiting factor with more threads. For benchmarks such as CG, FT and IS, the loops that are being parallelised are small and not very compute-intensive. However, the frequency with which they must be checked for dependences causes their performance to deteriorate at higher thread counts. This is demonstrated for the CG benchmark in Figure 6.17. As can be seen, at low thread counts ($<$12) most of the time is spent performing the loop execution, and the time spent checking for dependences is negligible. Beyond this point the time spent in execution continues to decrease, but the check time increases at a faster rate, preventing further speedup from being achieved. It is noticeable that the check time increases linearly with respect to the number of threads. Page table reset times are negligible for

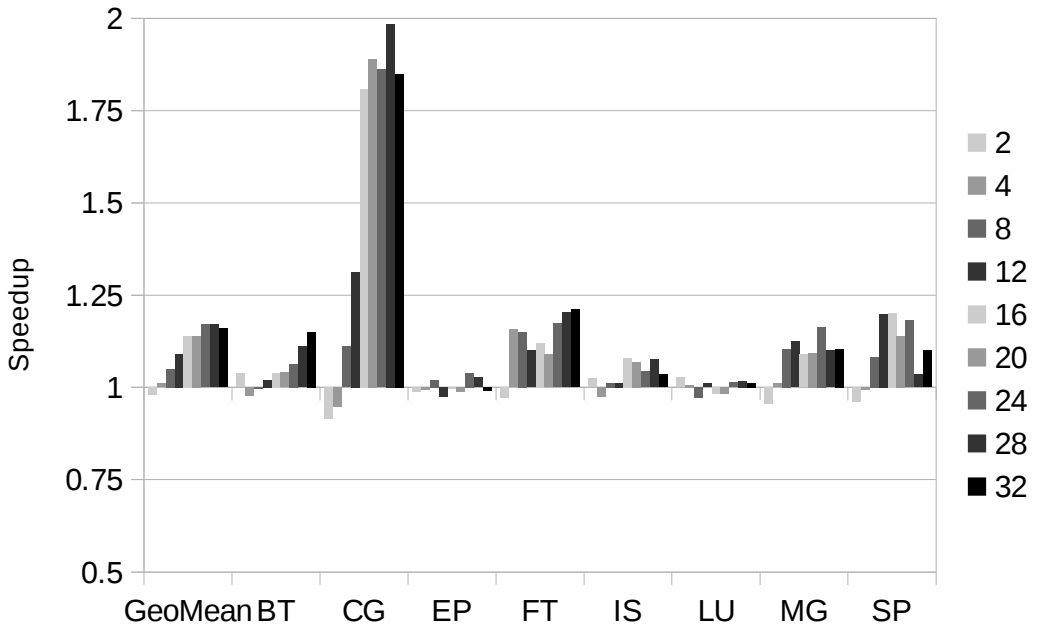


Figure 6.19: Speedup of reduction tree detection scheme compared to simple distributed scheme

all executions.

6.7.4 Reduction-Tree Detection Scheme

Figure 6.18 provides the speedups for each benchmark when using the reduction-tree detection scheme. The scheme managed to obtain up to 22.30× with a geometric mean of 2.50× speedup compared to sequential execution with 32 threads (1.82× excluding EP). As with the simple distributed scheme, BT, EP and MG continue to scale with the addition of more threads; however now FT also continues to scale. CG, IS and LU still peak at a lower thread count suggesting that the reduction scheme has the same problem with longer checks at higher thread counts. If the results of the EP benchmark are excluded the average speedup peaks again at 20 threads with a speedup of 1.86×. Finally SP also suffers from the same excessive check times using the reduction scheme.

6.7.4.1 Comparison to Simple Distributed Scheme

Figure 6.19 provides the speedup of the reduction tree detection scheme compared to the simple distributed scheme. As can be seen, at lower thread counts the two schemes

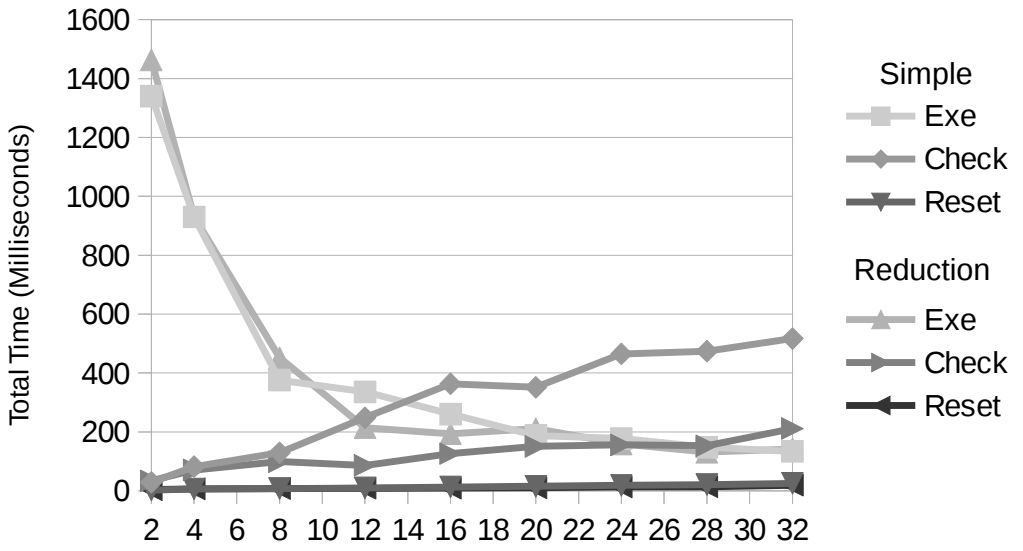


Figure 6.20: Execution time analysis of CG benchmark with both simple and reduction tree schemes

perform very similarly; occasionally the simple scheme performing slightly better albeit very nominally. For some benchmarks that do not perform many checks, such as EP and LU, this remains the case across all thread counts: the time spent checking is small in comparison to the execution of the benchmark. However, for every other benchmark executing with 8–16 threads and higher the reduction tree scheme becomes appreciably faster, with a geometric mean of 1.16× faster at 32 threads and up to 1.98× faster. For BT this speedup continues with additional threads. For IS and MG the speedup remains relatively constant at higher thread counts. Despite neither scheme managing to obtain a speedup for the SP benchmark it is clear that the reduction scheme does not slow down the execution as much. For the FT benchmark the speedups continue at higher thread counts although there is a lull at around 12 to 20 threads, due to the performance of the simple scheme hitting its peak.

Investigating the CG benchmark reveals that its performance using the reduction scheme is significantly better, executing almost twice as fast at higher thread counts. Figure 6.20 provides the execution times between execute, check and reset for the CG benchmark for both schemes across a range of threads. As can be seen for both schemes the execute times are roughly the same, and the reset times are negligible. However, looking at the checking times for each, there is a significant jump for the

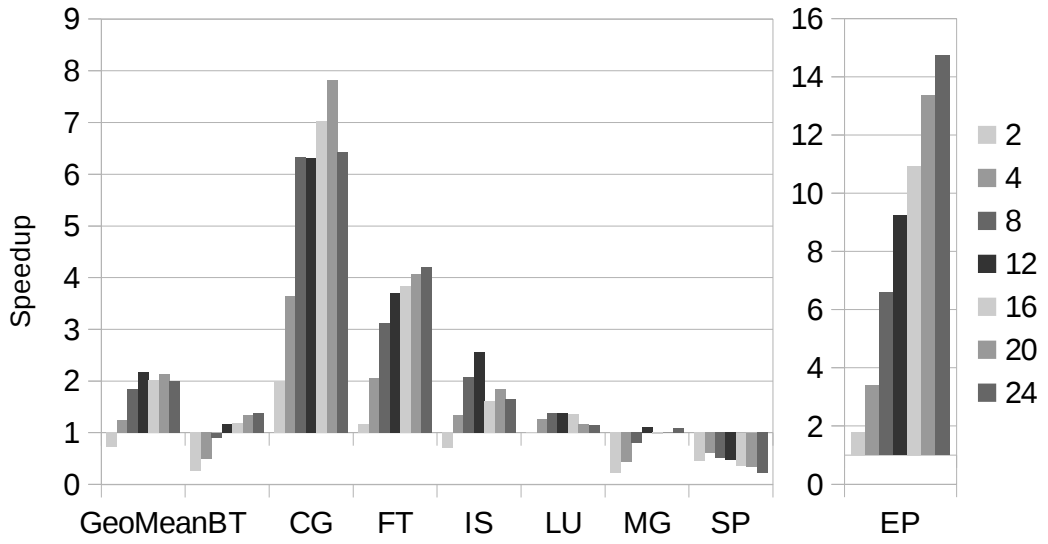


Figure 6.21: Speedup of hybrid CPU-GPU detection scheme

simple detection scheme above 8 threads. This is partially due to the architecture of the computer used for benchmarking, as beyond 8 threads execution must be performed across multiple processors. The reduction scheme also continues to grow as would be expected, but at a much slower rate than that of the paged scheme, allowing for continual improvements in execution times at higher thread levels.

6.7.5 Hybrid CPU-GPU Detection Scheme

Figure 6.21 shows the speedup of benchmarks using the CPU-GPU detection scheme. These benchmarks were evaluated on an alternative platform from the CPU-only based schemes due to the lack of availability of a GPU on the original platform, therefore each result was only able to be tested at up to 24 threads on a hyperthreaded platform. Full details of the platform are listed in Table 6.1. The sequential versions of each benchmark were re-executed on the GPU platform to give an accurate baseline comparison.

Overall the speedups obtained in this evaluation performed worse than the CPU-only evaluation. This is due to the lower thread count available and also the use of hyperthreading to obtain threads beyond 12. Hyperthreading is a form of threaded execution that alternates execution of two threads on the same core, sharing all caches and other

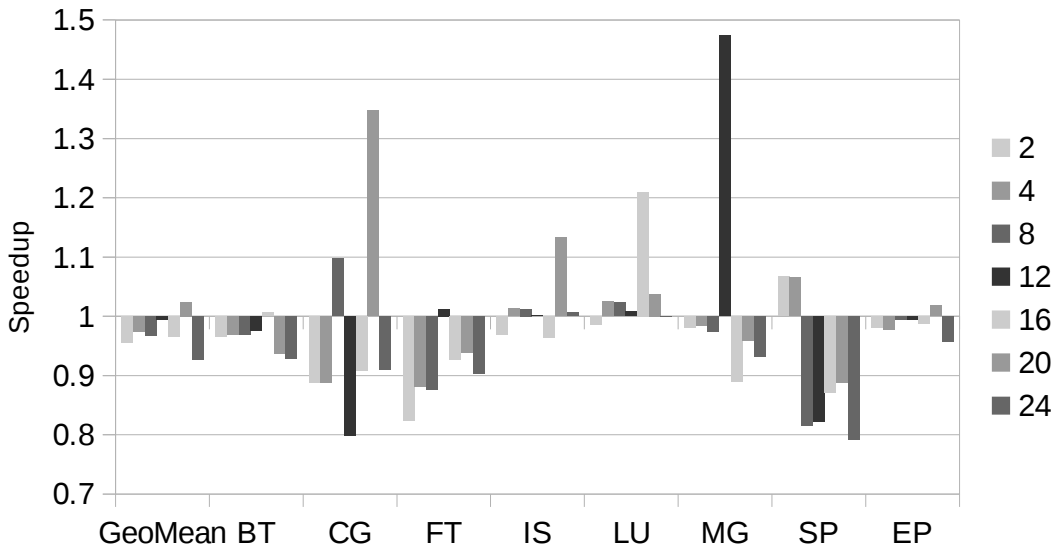


Figure 6.22: Speedup of Hybrid CPU-GPU Detection Scheme Compared to Reduction-Tree Scheme

resources of that core between each thread introducing additional restrictions on the performance of a thread, especially on the utilisation of CPU caches during memory tracing. Despite this, the evaluation was still able to obtain substantial speedups across a number of the benchmarks, up to 14.74× with a geometric mean of 1.99× compared to sequential when executing at 24 threads. Similar to the CPU models both BT and FT were able to achieve further performance increases with the addition of more threads, while IS, LU and MG peaked at lower thread counts. In this case they peaked at 12 threads, at the limit of one thread per CPU core. Unfortunately the MG benchmark was only able to obtain a small speedup 1.10×: as with the CPU-only versions, the use of hyperthreading made it impossible to obtain speedups beyond 12 threads. The CG benchmark continued to achieve performance increases at higher thread counts, but lost some performance at the highest count (24 threads). This is likely due to the fact that communications with the GPU were performed in a separate thread, thus influencing the performance and timings of the speculative threads. As with the CPU-only versions the SP benchmark was not able to achieve a speedup.

To compare this to a CPU-only scheme, each benchmark was re-executed on the new platform using the reduction-tree scheme. The speedup obtained by the CPU-GPU scheme compared to the CPU-only scheme can be seen in Figure 6.22. As can be seen

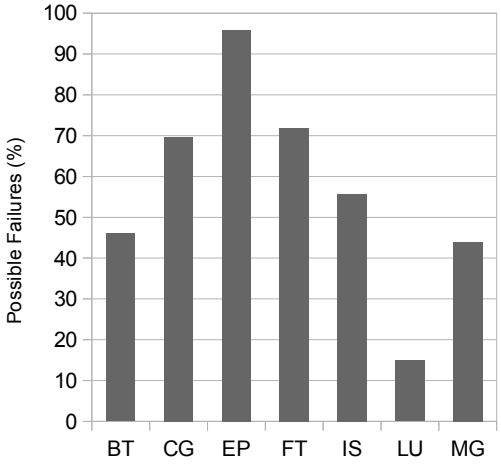


Figure 6.23: Percentage of *Kill-and-Restart* executions that can fail due to a dependence violation before overall performance is worse than sequential.

the hybrid CPU-GPU scheme performs worse than the CPU-only scheme in most, but not all cases. However this performance is roughly at $0.8\times$ speedup or higher, with a geometric mean of $0.96\times$ or higher. The worst example is the SP benchmark which performs poorly in all speculative cases. The poor performance is due to the costs associated with transferring data to the GPU, a situation that will likely improve as more modern APU designs, which integrate both CPU and GPU into the same wafer, become available. An alternative solution would be to fallback to a CPU-only scheme when the GPU is busy.

6.7.6 Dependence Violations

It is perhaps a little surprising that no dynamic dependence violations were detected in any of the above experiments. This indicates that the profile-guided parallelisation approach correctly identifies *probably-parallel* loops. While it is easy to construct a counter example, it suggests that many loops are genuinely parallel even though static analysis is unable to prove this. In fact, comparing the loops identified through analysis with those parallelised in the manually derived OPENMP reference implementation of the benchmarks confirm their equivalence (subject to insertion of speculation code).

Figure 6.23 displays the percentage of *kill-and-restart* executions (in their best configuration) that can fail due to a dependence violation before the overall performance becomes worse than sequential execution. The majority of benchmarks can have over

half of their executions fail due to a dependence violation and still obtain a performance increase. In fact, all but one of the benchmarks can have over 40%, or two executions in every five, fail due to a dependence violation and still obtain an overall performance increase. For the *EP* benchmark, 19 out of every 20 executions can fail due to a dependence violation and still obtain a performance increase.

One advantage of most benchmarks' performance peaking at fewer than the total quantity of threads is that this leaves additional computation units available for other uses. This provides us with enough additional resources to execute the safe, sequential version of the application in parallel using the *competitive scheduling* scheme. The lack of CPU resource contention allows for a worst-case execution time within 5%, on average, of the sequential execution time, being affected only by memory bandwidth limitations.

6.8 Conclusion

This chapter has demonstrated that often full SW-TLS schemes are unnecessary and that a more lightweight error-checking scheme can be used in its place. It has presented an automated way to extract and implement parallel execution through a combination of profiled-based parallelism detection and static analysis to detect variables that are the cause of many *may* dependences that restrict traditional auto-parallelisation techniques. Combined with error-checked execution this chapter has shown that substantial speedups can be achieved using this technique, with speedups of up to 22.53× and a geometric mean of 2.50× executing at 32-threads. By providing such speedups it has shown that even when dependences may exist in a program, but rarely surface, increased performance can still be obtained, with on average 2 in every 5 executions able to fail before performance becomes worse than sequential execution. In a *competitive scheduling* scenario this could be limited to a worst case of approximately sequential speed.

This chapter has also explored the use of auxiliary units, in this case a GPU, for the purpose of offloading some of the cost of error-checked execution. It has demonstrated that this technique, whilst performing slightly worse than the CPU-only, with a geometric mean of 0.96× speedup, will become viable with upcoming hardware improvements.

Now that GPU-based detection schemes have been shown to be a possible solution, the

next chapter introduces a completely GPU based speculation scheme, allowing both execution and dependence detection to be performed on a general purpose GPU.

Chapter 7

GPU-Based Speculation

GPUs have become ubiquitous in a wide range of computing devices and consumer electronics appliances. They provide a powerful resource for parallel processing and can deliver great performance improvements for suitably mapped algorithms. Realising this potential, however, is challenging due to the complexity of their programming.

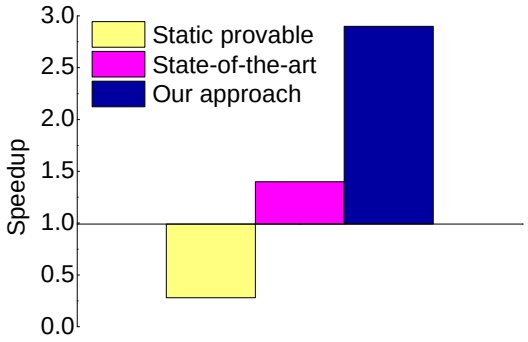
Auto-parallelisation technology can greatly reduce the barrier for GPU programming by automatically generating parallel code from sequential programs. However, auto-parallelisation targetting GPUs suffers from the same static analysis restrictions that affect CPU-targetted auto-parallelisation techniques. Fortunately, the same profiling techniques used to complement traditional static analysis in CPUs can also be applied to GPU targetted parallelisation, suffering from the same potentially unsafe parallel implementations. Speculative execution is the prime candidate to add safety to these implementations.

This chapter presents a solution that combines profile-guided parallelisation, OPENCL code generation and software thread-level speculation (SW-TLS) to exploit highly-likely parallelism on the GPU. This solution exploits static and profile-based dynamic dependence analysis to detect parallelism and to automatically generate parallel OPENCL code with in-place dependence checking. This solution exploits that parallel loop candidates are "almost always" genuinely parallel, but are not amenable to static analysis.

Section 7.1 provides an example situation that provides the motivation for speculative execution on a GPU. Section 7.2 presents the customised workflow necessary to perform speculative execution on a GPU along with the protection mechanism used to

```
1   void binvcrhs(double lhs[5][5],
2        double c[5][5], double r[5])
3   {
4     ...
5     lhs[1][1]=lhs[1][1]-coeff*lhs[0][1];
6       c[1][1] = c[1][1]-coeff*c[0][1];
7     ...
8   }
9   ...
10  void y_solve_cell() {
11    ...
12    for(j=1;j<grid_points[1]-1;j++){
13      for(k=1;k<grid_points[2]-1;k++){
14          binvcrhs(lhs[i][0][k][BB],
15                   lhs[i][0][k][CC],
16                   rhs[i][0][k]);
17      }
18    }
19  }
```

(a) Source code of an example loop



(b) Speedups obtained for the program

Figure 7.1: An example that static analysis fails to discover parallelism. No speedups were observed by only exploring statically provable parallelism. Profiling-based analysis, on the other hand, can provide additional information: no dependencies have been encountered in any trial run. By exploiting this information, the GPU can be used to execute both statically and probably parallel loops (with speculation support) and to achieve speedups rather than a slowdown. This approach gives a speedup of 2.9x which is 2 times faster than a speedup of 1.45x given by the state-of-the-art GPU speculation scheme.

ensure correct execution. Section 7.3 details the design of the data structures used to monitor memory accesses and the methods in which data dependences are detected. Section 7.4 discusses the code transformation and compilation framework. Section 7.5 with a thorough empirical evaluation in Section 7.6 and final conclusions in Section 7.7.

7.1 Motivation

Consider the code fragment in figure 7.1 (a). This loop is extracted from the BT benchmark from the sequential version of the NAS benchmark suite. While conservative, static analysis fails to exploit the parallelisation opportunity of this loop due to the

inter-procedural call to function `binvcrhs` at line 14. Here, an output dependence (that is, write-after-write) to array `lhs` has to be assumed. Without further information, this loop would have to be executed sequentially on the CPU (as it is too expensive to do so on the GPU). Although it is possible to execute some loops on the GPU where they are statically provable to be parallel, additional synchronisation will have to be introduced and communications between the sequential CPU and parallel GPU executions. The additional overhead, however, could be expensive and can outweigh the benefit of GPU parallel execution. In fact, as can be seen from Figure 7.1 (b), doing so leads to a slowdown of 3.6× over the sequential code on a NVIDIA GTX 580 platform described in Section 7.5.

Profile-based dependence analysis, on the other hand, provides the additional information that *no actual data dependence* inhibits parallelization for the given sample inputs. While it is not possible to prove the absence of data dependences for *every possible input*, the loop can be classified as a highly-likely parallel candidate. The loop can then be speculatively executed in parallel on the GPU with dependence violation checking together with a rollback scheme to ensure correctness if a true dependence violation is discovered at runtime. This is safe and potentially fast. As shown in Figure 7.1(b), a state-of-the-art GPU speculation scheme, Paragon [37], gives a speedup of 1.45× for this particular benchmark. Though the result of Paragon is encouraging, it can be further improved. Paragon requires a large buffer to record the speculative accessing addresses, which will be used in a separated dependence checking procedure to check the potential violations of speculative accesses. This, however, can result in expensive indirect memory accessing overhead on the GPU. It is preferable to avoid this overhead.

As described later in this chapter, the novel *in-place* dependence checking approach does not require a buffer to store the speculative accessed addresses. It gives a speedup of 2.9× — twice as fast as Paragon. With a novel dependence checking scheme, a compiler framework is built to automatically generate parallel OpenCL code from sequential code using dependence profiling information and without user interaction, allowing the exploitation of GPU parallelism for legacy code that is highly likely to be parallel.

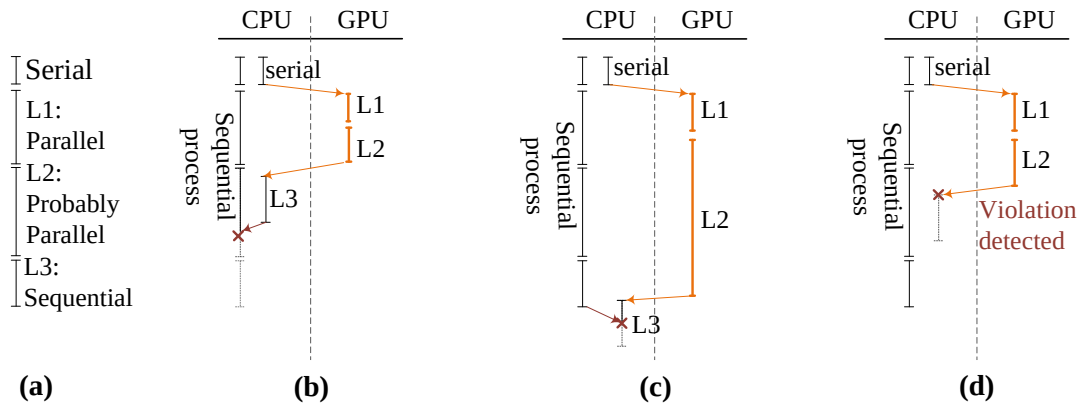


Figure 7.2: Three different parallel execution scenarios for the sequential program shown in (a): speculative execution runs faster with no conflict (b); sequential execution runs faster (c); violations are found for speculative execution (d).

7.2 Execution Workflow

Ideally all suitable parallel executions would be performed solely on the GPU. However, by their nature, speculative parallel executions can fail despite prior dependence profiling. Due to the memory architecture and traditional execution processes that a GPU provides, it is infeasible to provide version control and rollback mechanisms on the device itself. Doing so would cause excessive uncoalesced accesses to global memory, and create code that is unreasonably divergent between each thread's path of execution. Instead an alternative execution workflow must be used to provide a suitable version control system.

Similar to that of Paragon, a *competitive scheduling* scheme is used: a sequential version of the program is executed simultaneously alongside the parallelised program on a spare core of the host CPU. If a dependence violation is reported, the speculative parallel execution is simply aborted, with the result produced by the safe, sequential run as the output of the program. As an added benefit, competitive scheduling caps the maximum execution time to that of the sequential program.

Figure 7.2 depicts the competitive scheduling scheme. This example contains three loops: a statically proven parallel loop $L1$, a probably parallel loop $L2$, and a statically proven sequential loop $L3$. In this scheme, loops $L1$ and $L2$ will be executed on the GPU and the sequential loop $L3$ will be executed on the host CPU. There are three possible scenarios. If the speculative version finishes first and does not observe any dependence violations, it terminates the sequential version (Figure 7.2 (b)). If

the sequential version finishes first, it will abort the parallel speculative version (Figure 7.2 (c)). Finally, if the speculative version detects a dependence violation, it aborts and the sequential version will eventually finish successfully (Figure 7.2 (d)).

7.3 Speculative GPU Execution

This sections presents the GPU-based speculative execution process. The scheme presented is inspired by a CPU-based SW-TLS scheme, SPLIP [30]. Proposed is an *eager*, *in-place* dependence checking scheme for GPUs. Checking only needs to be applied to speculative memory references in probably parallel loops. Statically-provable parallel loops require no runtime checking at all. Dependences are checked *in-place* and on-the-fly, and any violation is reported to the control thread on the CPU.

The design and use of the data structures required to log memory traces are detailed in Section 7.3.1. Details of how dependence violations are detected and the types of violation that are supported is presented in Section 7.3.2. Finally a comparison to another GPU detection scheme is presented in Section 7.3.3.

7.3.1 Speculative Data Structures

Due to the *eager* nature of this design, *centralised* data structures are used to record each memory trace. However, unlike traditional CPU execution, GPU execution prevents the use of several programming paradigms that must normally be considered during speculative execution. Notably, the use of pointer arithmetic is heavily restricted, disallowing the possibility of overlapping arrays and pointer aliasing. This allows the use of multiple, *per-variable* memory traces without the risk of cross-trace dependence violations going undetected.

As discussed in Section 2.3.6.3, *per-variable* traces can dramatically reduce the size of speculative data structures and allow for simpler and more customisable address hashing functions. In the evaluation of this work, the compactness of each trace allowed a perfect hash to be used, preventing the possibility of false dependences.

The trace structure used is similar to that used by SPLIP. Each trace consists of two centralised arrays of counters, a read log and a write log, with each counter storing the highest iteration to access that location at a given moment. Figure 7.3

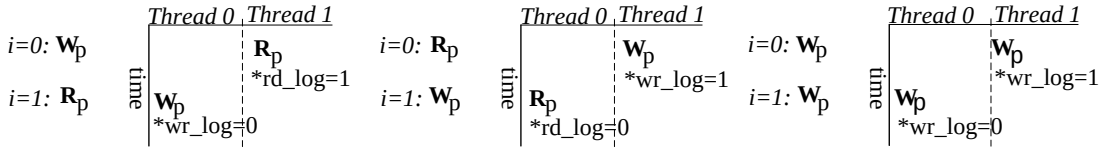
```
1  double specLD_double(__global double *a, __global int *wr_log,
2         __global int *rd_log, int iter_id, __global int *flag)
3  {
4    double value;
5    atom_max(rd_log, iter_id);
6    value = a[0];
7    if (*wr_log > iter_id) /*Condition 1*/
8          *flag = FAIL;
9    return value;
10 }
11
12 double specST_double(__global double *a, __global int *wr_log,
13      __global int *rd_log, int iter_id, __global int *flag, double value)
14 {
15    atom_max(wr_log, iter_id);
16    if (*wr_log > iter_id) { /*Condition 2*/
17        *flag = FAIL;
18    }
19    a[0] = value;
20    if (*rd_log > iter_id) { /*Condition 3*/
21        *flag = FAIL;
22    }
23    return value;
24 }
```

Figure 7.3: The OpenCL implementation of a speculative load and store. Dependence checking is combined with speculative loads and stores.

shows the OPENCL implementation of speculative load and store operations. For each speculatively-accessed address, the corresponding entry in either the read or write log is updated, that is, the `rd_log` and `wr_log` variables in Figure 7.3. As OPENCL does not support barriers for GPU threads across work groups, the `atom_max` operation provided by OPENCL was used to ensure only the *highest* iteration ID is stored in the log (lines 5 and 15). The value of each log entry monotonically increases over time.

7.3.2 Violation Detection

As discussed, dependence detection is performed *eagerly*. This takes place during speculative loads and stores, so checks for dependence violations only need to be performed for addresses that are actually accessed at runtime.



(a) flow dependence (b) anti dependence (c) output dependence

Figure 7.4: Three cross-iteration dependences and the possible runtime violations due to GPU thread scheduling. All three violations can be successfully detected by the dependence checking scheme with the read (`rd_log`) and write (`wr_log`) buffers as shown in Figure 7.3.

7.3.2.1 Speculative Load

A speculative load is successful if there has been no speculative store to the same memory location by a GPU thread that executes a later loop iteration. This condition is checked in line 7. If the memory location has been written to by a later iteration (`*wr_log > iter_id`), a violation will be reported (line 8).

7.3.2.2 Speculative Store

Conversely, a speculative store is successful as long as there have been no speculative accesses (either loads or stores) to the same address by later iterations. This condition is checked in lines 16 and 20. If a later iteration has already written to the same location (`*wr_log > iter_id`) or read from it (`*rd_log > iter_id`), a violation is detected.

An added benefit of the *eager* design becomes apparent if cross-iteration dependent accesses are executed in the correct sequential order by virtue of the GPU thread scheduler: this situation is correctly handled, without a violation being flagged.

The parallelisation scheme proposed maps each loop iteration to an OPENCL work item to be executed by one GPU thread. Hence, no dependence violations are possible within a single iteration. Cross-iteration dependence violations, on the other hand, *are* possible, due to the arbitrary order of thread scheduling on the GPU. In this case, Figure 7.4 enumerates all three possible cross-iteration violations. Shown is the sequential dependence of two consecutive iterations that must be respected and the potential violation due to GPU thread scheduling.

7.3.2.3 Flow Dependence

Figure 7.4 (a) illustrates a violation of a flow dependence (read after write), where the use of `p` in iteration 1 happens before `p` is updated by thread 0, which executes iteration 0. This violation will be detected in function `specST`. In this case, `*rd_log == 1` and `iter_id == 0` and Condition 3 (line 18) of Figure 7.3 holds, such that a violation will be reported.

7.3.2.4 Anti Dependence

In Figure 7.4 (b), the use of `p` happens after it has been updated by the a later iteration. This causes an anti-dependence (write after read) violation, which will be captured by function `specLD`. In this case, `*wr_log == 1` and `iter_id == 0` and Condition 1 (line 7) of Figure 7.3 holds, such that a violation will be reported.

7.3.2.5 Output Dependence

Figure 7.4 (c) is an output dependence (write after write) violation. After thread 1 has updated `p`, this memory location is overwritten by thread 0, which executes a previous iteration. In this case, `*wr_log == 1` and `iter_id == 0` and Condition 2 (line 16) in Figure 7.3 holds, such that a violation will be reported.

7.3.3 Comparison to Other Approaches

The proposed speculative checking scheme has several advantages when compared to other state-of-the-art GPU thread level speculative schemes, such as Paragon [37]. Unlike Paragon, this scheme does not explicitly record addresses of speculative memory accesses. Instead checking is performed on-the-fly as an integral part of the speculative accesses. As such, this scheme does not have the indirect memory access overhead resulting from the address bookkeeping buffer, a problem which hampers Paragon's performance. This scheme is particularly well suited for sparse data applications (e.g. using sparse matrices) where only a small proportion of the total index space is accessed by the program. Unlike Paragon, load and store logs (`rd_log` and `wr_log`) can be reused between multiple speculative kernels without the need for clearing them in

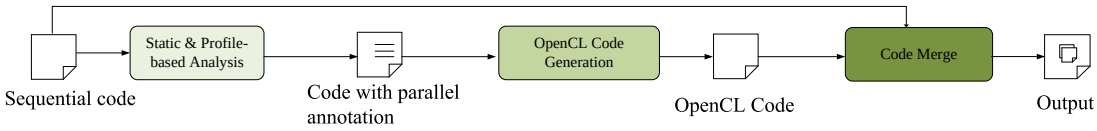


Figure 7.5: The compiler framework first uses static and profile-based analysis to identify parallel candidates. Those parallel candidates are then translated into OpenCL kernels. Dependence checking code is added to perform dependence checking for those candidates that cannot be statically proven to be parallelisable but no dependence violation was discovered during profiling. Finally, the generated parallel OpenCL program is merged with the original sequential program as output.

between. Finally, Paragon uses a naive violation-detection scheme where an output dependence violation will be reported if there is more than one write to the same memory address. This naive scheme may cause false positives (that is, a successful speculative execution is reported as a violation) when an address has been updated multiple times within the same loop iteration or a write dependence is honoured. By contrast, the proposed precise violation detection scheme is *exact* and does not suffer from this problem.

7.4 Compilation and Code Transformations

Figure 7.5 depicts the compilation framework for the proposed scheme. The compiler uses three steps to generate parallel GPU code: parallelism detection, OPENCL code generation and code merging.

7.4.1 Parallelism Detection

This work targets loop-level parallelism. In particular, static analysis is used to separate definitely-sequential and definitely-parallel loops from other loops, which may or may not be parallel. For these possibly-parallel loops the scheme relies on dependence profiling [44, 47] to extract those loops which are *probably parallel*. These loops are marked as probably parallel if no cross-iteration dependences have been observed during any profiled execution using different data inputs. These loops are candidates for speculative parallel execution. The output of this stage is a program

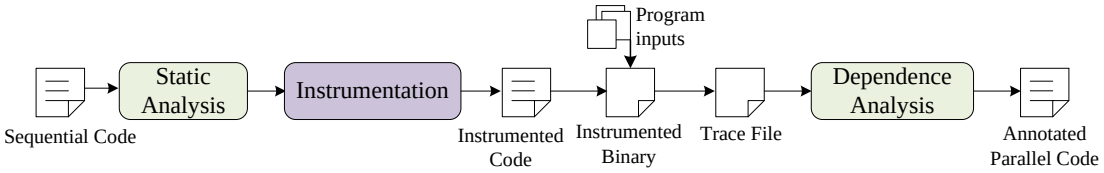


Figure 7.6: The process of profile-based dependence analysis. The compiler only uses profile-guided analysis for code regions where static analysis has bailed out.

with OPENMP-like annotations to parallel and probably-parallel loops, which include privatisable variables.

Figure 7.6 illustrates the hybrid static and dynamic parallelism detection approach. Static analysis uses a customised memory dependence analysis path from LLVM v3.4. Profile-guided analysis with similar capabilities to those of [44] is then performed, except instrumenting only those memory operations that previous static analysis could not resolve with certainty. The original, sequential application is recompiled and executed with several different inputs to generate traces of memory operations. Loop traces are further analysed to determine if data dependences occurred during execution. Any loop that does not contain cross-iteration data dependences is then marked as *probably parallel*. Additionally, traces can be used to support static reduction recognition.

7.4.1.1 Speculative Variables

Tracking speculative memory accesses is expensive, so it is desirable that only those accesses that could potentially cause a dependence violation are tracked. Here static analysis generates a list of variables that require speculative tracking — those which are subject to *may*-dependences. In particular, accesses to read-only and thread-private are variables *not* tracked. For the remaining speculative accesses suitable wrappers are inserted which invoke the appropriate checking functions.

7.4.2 OpenCL Code Generation

The annotated program is passed to an OPENCL code generator, which automatically converts data-parallel loops and parallel reduction loops into OPENCL kernels. Each data-parallel loop is translated to a separate kernel using the OPENCL APIs, where each

	Intel CPU	NVIDIA GPU
Model	Core i7	GTX 580
Core Clock	3.6 GHz	1544 MHz
Core Count	6 (12 with HT)	512
Memory	12 GB	1.5 GB
Peak Performance	122 GFLOPS	1581 GFLOPS

Table 7.1: Hardware platform

iterator of the loop is replaced by a global work-item ID. Checking code is added to speculative references, which may lead to a dependence violation in probably-parallel loops. Figure 7.7 provides a simplified OpenCL-based code for the statically undecidable parallel loop shown in Figure 7.1.

7.4.3 Code Merging

The last compilation stage merges the generated parallel OpenCL code with the original, sequential program into a single program. As such, the output program consists of both the original, safe implementation in addition to the generated OpenCL parallel code.

7.5 Experimental Setup

7.5.1 Platform

This approach has been evaluated on a CPU-GPU mixed system with an Intel Core i7 CPU and an NVIDIA GTX 580 GPU. The system runs with openSUSE 12.3 with Linux kernel 3.7.10. Table 7.1 gives detailed information of the platform.

7.5.2 Benchmarks

The NAS benchmark *sequential* v.2.3 suite was used, for which manually parallelised CPU and GPU implementations are available. To parallelise the code, a profiling-based

```
1  void binvcrhs_spec(__global double (*lhs)[5],
2                     __global int (*rd_log_lhs)[5],
3                     __global int (*wr_log_lhs)[5],
4                     ...,
5                     __global int* spec_flag,
6                     __global int iter_id)
7  {
8    ...
9    _rval_0 = specLD_double(&lhs[1][1],
10                           &wr_log_lhs[1][1],&rd_log_lhs[1][1],
11                           iter_id,spec_flag);
12
13   _rval_1 = specLD_double(&lhs[0][1], ...);
14
15   //speculatively store the result to lhs[1][1]
16   specST_double((_rval_0-coeff*_rval_1), &lhs[1][1],
17                 &wr_log_lhs[1][1], &rd_log_lhs[1][1],
18                 iter_id,spec_flag);
19   ...
20 }
21
22 __kernel void y_solve_cell_L0 (...)
23 {
24     ...
25     iter_id = get_global_id(1) * get_global_size(0)
26               + get_global_id(0) + init_iter_num;
27     ...
28     binvcrhs_spec(lhs, rd_log_lhs, wr_log_lhs,
29                   lhs, rd_log_lhs, wr_log_lhs,
30                   rhs, rd_log_lhs, wr_log_lhs,
31                   spc_flag, iter_id);
32 }
```

Figure 7.7: A simplified OpenCL-based code for the statically undecidable parallel loop shown in Figure 7.1. A speculative version of the original function `binvcrhs` is generated in which every access to the speculative variable `lhs` is replaced with a speculative load/store operation.

auto-parallelisation tool was used to analyze the data dependences and generate parallel OpenCL code. The tool uses speculative checking to parallelise loops that are found to be parallelisable during profiling but cannot be statically proved parallelisable. For all loops that can be statically proven to be safe to parallelise, the tool parallelises them straightforwardly. The compiler parallelises up to three levels of nested loops, to create as many GPU threads as possible. Whenever possible, CPU-GPU communication and synchronisation was avoided by running a parallel loop on the GPU. Loops that accounted for less than 1% of the whole-program execution time were also avoided unless there was a consecutive parallel or probably parallel loop candidate after it. This was so that a CPU-GPU synchronisation point could also be avoided.

7.5.3 Compiler and Evaluation Runs

All programs were compiled using GCC 4.4.7 with the `-O3` option. Each experiment was repeated five times and the geometric mean execution time was recorded. All the benchmarks were profiled using the smallest input provided by the benchmark (class S) and evaluated with a larger input class (class A).

7.5.4 Comparison

This approach is evaluated against Paragon [37], the closest competitor. In Paragon, probably-parallel loops are discovered at program runtime by profiling loops that are statically undecidable. However, experimentation revealed that doing so is very expensive. To provide a fair comparison, the profiling stage was performed offline and Paragon was provided with the same probably-parallel code, forcing speculation to occur on exactly the same loops for each approach. Therefore, only the efficiency of speculation rather than accuracy of parallelism discovery and profiling overhead was evaluated. The Paragon scheme relies on OpenCL code generation. Again, the same OpenCL code generator was used to provide a fair evaluation. In addition to Paragon, a comparison of the approach is provided to two manually parallelised implementations of the NAS benchmark suite: an OPENMP version and an OPENCL implementation (SNU NPB [40]). Both versions were implemented by independent programmers. The two manual implementations provide a good estimation of the upper-bound performance achievable with the help of user assistance.

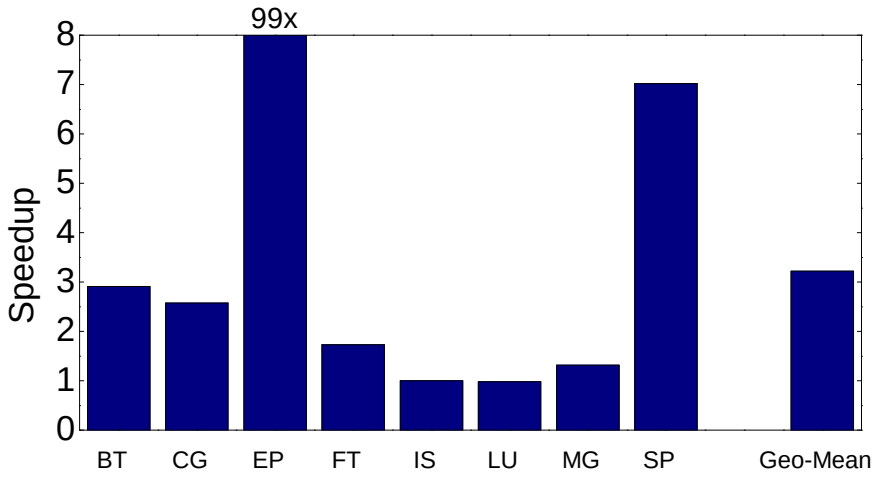


Figure 7.8: Speedups over the sequential execution of the approach. A geometric mean speedup of 3.2× was acheived and parallel execution is never significantly slower than sequential execution.

7.6 Empirical Evaluation

In this section the approach is evaluated against the sequential baseline followed by a comparison of the approach to a scheme that only parallelises statically decidable loops on GPUs. This is followed by comparisons to a state-of-the-art GPU speculation scheme and the manually parallelised implementations. Finally, a closer look is taken at the limitations of static analysis and the speculation overhead, with a discussion of dependence violations.

7.6.1 Overall Results

Figure 7.8 shows the speedups achieved by the scheme. The performance numbers presented are speedups over the sequential execution on the CPU. The scheme achieves a geometric mean speedup of 3.2×. Furthermore, by employing competitive-scheduling the scheme has never significantly slowed down the program.

As can be seen from Figure 7.8, exploiting GPU parallelism for probably-parallel loops can realise a great performance improvement. This is exemplified by the embarrassingly parallel benchmark EP, where a speedup of 99× was observed. Parallel GPU execution can be of benefit for other benchmarks too. For benchmarks BT, CG and SP, a speedup of at least 2.6× and up to 7× was achieved. For benchmarks FT and MG,

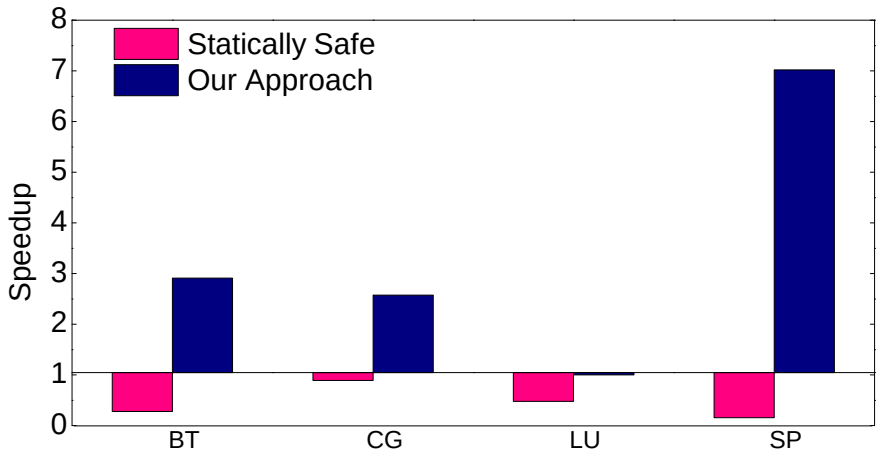


Figure 7.9: Comparisons of the statically parallelised version and the in-place GPU speculation scheme. The in-place scheme achieves substantial speedups on most benchmarks, with the static version acheiving only slowdowns.

only modest speedups were achieved due to the available parallelism and cost of speculation. For benchmarks LU and IS, no speedups were observed on the platform. For LU, a new algorithm is required to obtain improved performance on the GPU [40, 12]. For IS, the parallel loop only accounts for 27% of the sequential execution and it is not worth parallelising it on the GPU. Nonetheless, the competitive-scheduling scheme caps the execution time to the time of the sequential run if the parallel GPU execution is not profitable.

7.6.2 Comparison with the Statically Safe Approach

Presented is a comparison of the new approach to a conservative approach that only parallelises those statically proven parallel loops on the GPU, running the rest sequentially on the CPU. Obviously, no speculation is needed for such a scheme but data transfers and synchronisation are required to synchronise between the CPU and the GPU threads.

Figure 7.9 compares the new approach with such a statically safe scheme. Here, some of the benchmarks are omitted because static analysis fails to discover any parallelism in them. As can be seen from this figure, no speedups were observed for the conservative, safe scheme. This is due to the communication and synchronisation overhead associated with the switch between the CPU and GPU executions, where shared variables have to be synchronized among the two devices. These costs outweigh the bene-

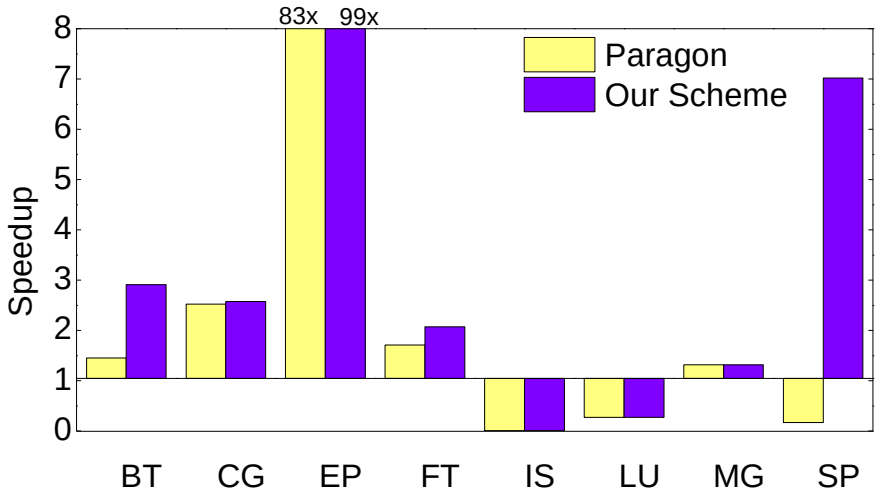


Figure 7.10: Comparison of Paragon and the in-place GPU speculation scheme. The in-place approach achieves higher speedups on more benchmarks when compared to Paragon.

fit of GPU parallel executions. The new approach, by contrast, avoids these overheads by running two consecutive static and probably-parallel loops on the GPU so that all data can be kept on the GPU and otherwise expensive CPU-GPU data transfers can be avoided. Unlike the disappointing results of the static scheme, the new profile-based GPU speculation scheme is able to achieve speedups for all the four programs except LU (where a change of algorithms is required to achieve speedups on the GPU [40]).

Overall, static parallelisation technology is too conservative to exploit GPU parallelism despite the abundant available parallelism for the majority of benchmarks. By contrast, the new approach outperforms the static parallelisation approach by a factor of seven.

7.6.3 Comparison with Paragon

Figure 7.10 compares the new GPU speculation scheme with Paragon. The performance achieved by co-running the sequential code has been factored out so that the focus is solely on the quality of the GPU speculation scheme. Note that the same OPENCL code optimizations were applied to both approaches; therefore, the performance variations are mainly down to the difference of the speculation schemes.

This figure clearly demonstrates the advantages of the new approach. As can be seen from this diagram, the overhead of Paragon can be significant for some benchmarks.

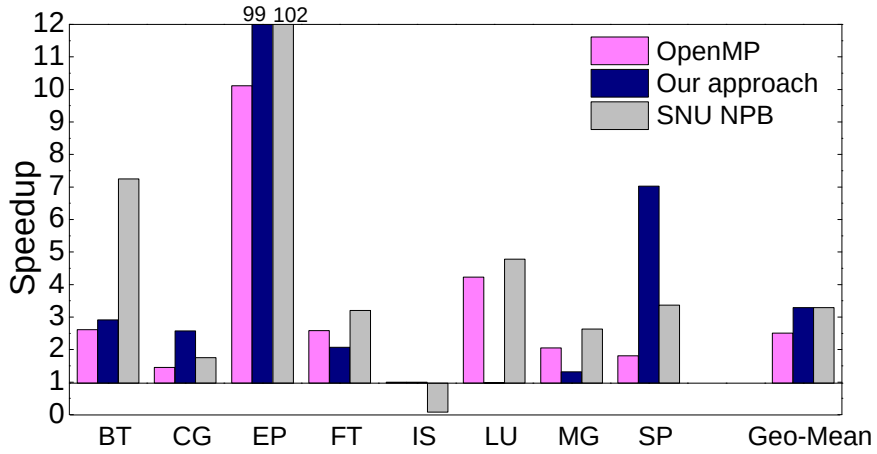


Figure 7.11: Performance of the manual OPENMP and OPENCL implementation of the NAS benchmark suite and our automatically generated parallelised code.

For example, Paragon is not able to achieve speedups for SP while the new approach gives a speedup of over 7×. For this benchmark, the indirect memory accessing and initialization overhead of Paragon clearly outweighs the benefit of GPU parallel execution. Besides SP, the new scheme also outperforms Paragon on benchmarks BT and FT, with a speedup up to 2 times higher. For benchmarks CG, EP and MG, speculative checking only needs to be performed on a few speculative variables and both approaches deliver similar performance. Finally, for benchmarks IS and LU, neither of the two schemes achieve performance improvement due to the restriction of the program and the GPU architecture as explained in section 7.6.1. Overall, the new scheme outperforms Paragon: it achieves higher speedups whenever it is profitable to exploit GPU parallelism.

7.6.4 Comparison to Manually Parallelized Code

Also provided is a comparison of the new approach to two manually parallelised implementations developed by independent programmers: (1) the OpenMP version of the NAS benchmark suite [2] for the CPU and (2) SNU NPB [40], an OPENCL implementation of the NAS benchmark suite for the GPU. The SNU NPB provides a good estimation of the upper-bound performance that the new GPU speculation scheme can achieve. The results are shown in Figure 7.11.

As can be seen from this diagram, exploitation of GPU parallelism for highly-likely parallel code can be beneficial. Example benchmarks include BT, CG, EP and SP

where GPU execution significantly outperforms the OpenMP CPU execution by a factor of up to 10. It is not surprising that a manually parallelised GPU implementation without speculation overhead outperforms the new automated scheme, but the new approach is able to achieve a level of performance close to the manual implementation. For benchmarks CG and SP, the approach even outperforms the manual GPU implementation with advanced GPU memory optimizations such as dynamic index reordering applied by the OPENCL code translator [12]. For benchmarks FT and MG, the new approach is not as good as the OpenMP implementation. This is restricted by the programs themselves as the CPU-GPU communications are relatively high compared to computation. This can be seen from the fact the manually parallelised GPU code only outperforms the OpenMP CPU code by a small margin. For the benchmark LU, the algorithm in the sequential code has to be changed to a hyperplane one to achive speedups on the GPU [40]. This is of course out of the scope of the automated approach. Finally, for IS, none of the three parallel versions can gain speedups because the execution time of this program is dominated by serial code.

Overall the new automatic approach performs well. The 3.2× geometric mean speedup achieved by this approach is very close to the 3.3× speedup of the manually parallelised OPENCL implementation. Moreover, the approach also outperforms the OPENMP implementation on the majority of the benchmarks by exploiting GPU parallelism.

7.6.5 Analysis

7.6.5.1 Limitation of Static Analysis

Table 7.2 shows the number of parallelised loops of the OpenMP implementation and, among those, how many are statically decidable and undecidable. For benchmark CG, a considerable number of the parallelised loops are statically decidable. However, for most of the programs, merely relying on static analysis is not enough to exploit program parallelism, which actually misses a significant amount of parallel opportunities. For example, for benchmarks EP, FT and IS, static analysis fails to detect any of the manually parallelised loops. This failure of static analysis to exploit parallelism is particularly telling for the EP benchmark, where a speedup of 90× is available. Dependence profiling information, on the other hand, can provide additional information, enabling the discovery of those parallel opportunities. By contrast to static analysis,

Benchmark	Manual	Statically Decidable	Statically Undecidable
BT	54	23	31
CG	19	17	2
EP	1	0	1
FT	6	0	6
IS	1	0	1
LU	29	12	17
MG	12	5	7
SP	70	41	29

Table 7.2: Numbers of statically-decidable and -undecidable parallel loops of the manual OpenMP implementation.

the new hybrid static and dynamic parallelism detection scheme identifies all the parallel loops specified in the OpenMP implementation. This table shows that profile-based analysis is a powerful technique that allows the discovery of parallelism for legacy code that is highly likely to be parallel.

7.6.5.2 Speculation Costs

Figure 7.12 shows the overhead of the speculation overhead for each benchmark. In this diagram, the program runtime is broken down into two parts: speculation overhead and non-speculative GPU parallel execution. The two breakdowns are shown as the percentage to the overall program runtime. As can be seen from this diagram, the speculation overhead varies from one program to the other. Depending on the number of probably-parallel loops and the frequency of speculative accesses, the overhead varies from 60% to 15% relative to the whole-program execution time. For some benchmarks, such as CG, FT and SP, the speculation overhead is relatively low, around 20%. This is because speculation only needs to be applied on a few arrays. For benchmark LU, the program execution time is dominated by the synchronisation and communication overhead due to the restriction of the program algorithm and thus the speculation overhead is not significant. For benchmarks IS, MG and BT, the overhead is more than 30% of the whole-program execution time, because of the high frequency of speculatively accessed variables. Particularly, benchmark BT has the highest speculation overhead, accounting for 60% of the total program execution time. For this benchmark, 31 out

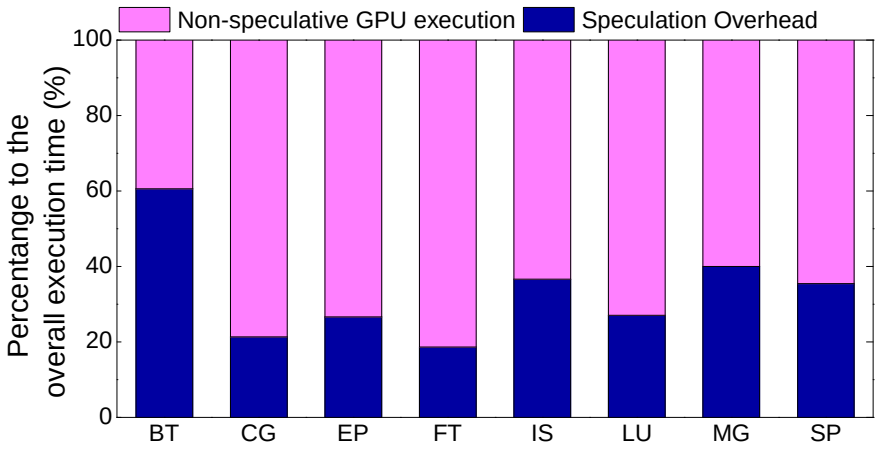


Figure 7.12: Speculation overhead compared to the unsafe parallel execution without speculation on the GPU.

of the 54 parallel loops cannot be statically determined, and speculation must be performed on those statically undecidable loops. Despite the speculation overhead, the new approach is still able to achieve a speedup of 2.9× as opposed to the 3.6× *slowdown* of a static approach (see Section 7.1). On average, the speculation overhead is 28% across all benchmarks.

7.6.5.2.1 Dependence Violation It is perhaps a little surprising that no dynamic dependence violations were detected in any of the above experiments. This indicates that the profile-guided parallelisation approach correctly identifies probably parallel loops. While it is easy to construct a counter example, it suggests that many loops are genuinely parallel even though static analysis is unable to prove this. In fact, a have comparison of the loops identified by this analysis with those parallelised in the manually derived OPENMP reference implementation of the benchmarks confirmes their equivalence (subject to insertion of speculation code).

7.7 Conclusion

This chapter has presented a holistic approach to exploit parallelism for highly-likely parallel legacy code on commodity GPUs. Building on prior work on profile-guided parallelization, a novel GPU-based speculation scheme to provide correctness guarantees for probably parallel loops has been proposed. This scheme discards expensive

check-pointing for rollback that is not suitable for GPUs, and instead provides faster dependence checking for speculative parallel execution regions, which are identified as probably parallel by the profile-guided analysis. This novel approach allows dependence checking to occur in-place with speculative access operations. Thus checking only needs to be performed only on those addresses where speculative accesses actually take place. This approach has been evaluated on benchmarks that are rich in parallelism but hard to parallelise using traditional static analyses. By exploiting GPU parallel execution, the expensive overhead that otherwise would be required for serial CPU executions was avoided. This chapter has demonstrated the effectiveness of the in-place GPU speculation scheme by comparing it to a state-of-the-art GPU-based speculation scheme. Experimental results show that the new technique outperforms the state-of-the-art by a factor of 1.45. This translates to 99% of the performance of a manual OPENCL implementation without speculation overhead where the probably-parallel loops have been manually verified.

Chapter 8

Conclusion

This thesis has investigated the use of software thread-level speculation for automated parallelisation in several ways: by furthering the research and techniques available to fully automate the speculative workflow and to contribute to the performance of speculative execution by introducing a number of new speculation techniques. Particularly the process of implementing speculative execution has always been a manual and relatively time-intensive task, requiring detailed analysis of each program to obtain performance improvements.

This work has presented means to automatically select suitable probably parallel regions, select an appropriate speculation policy and translate existing legacy sequential code to execute safely in parallel. It has also investigated the use of general purpose GPUs for speculative execution providing additional possible platforms where performance increases can be obtained, and also further work into creating extremely lightweight speculation schemes.

The rest of this chapter is structured as follows: Section 8.1 presents a summary of the contributions made by this thesis with an analysis of the experimental results it has provided. Section 8.2 provides a short analysis of the work with a discussion of potential areas of future research.

8.1 Contributions

8.1.1 Automated Policy Selection

Chapter 5 introduced a fully automated technique to select and exploit appropriate parallel opportunities that would provide suitable performance increases when executed speculatively. This is performed by extracting metrics from a loop as the input to a machine-learning-based configuration tool to decide whether a loop should be speculated over and if so, which speculative policy would be the most profitable. This technique requires an expensive *off-line* training period to collect the required data points for the configuration tool, however this procedure is fully automatable and is only required to be performed once for a given architecture.

Evaluation of this technique yielded substantial speedups across a number of benchmarks with a geometric mean of 1.64× and up to 7.75× over sequential on an 8-core AMD Opteron machine. A number of different machine-learning-based prediction techniques were tested that resulted in 89% of the selected policies executing at a speed of sequential or better. On average approximately 74% of the speedup obtainable through manual selection and configuration was achieved.

8.1.2 Lightweight Error Checking

Chapter 6 investigated the concept that many SW-TLS schemes are very heavyweight when used with modern profile-based auto-parallelisation techniques. In cases where profiling reveals loops that are almost certainly parallel these schemes prioritise recovery from a dependence violation above that of actual execution. This thesis showed that in these cases traditional version control used in speculative schemes was unnecessary, and that instead a very lightweight error checking scheme could instead be used introducing the potential of an extremely expensive rollback scenario.

To minimise the risk of the rollback scenario a completely accurate access tracking technique was introduced that dynamically allocated memory to the trace as necessary reducing the overall memory footprint of the technique resulting in traces that used 3 percent or less than existing schemes across the same address space.

Several detection schemes were introduced; several CPU-only schemes and one hybrid

CPU-GPU scheme. This thesis showed that using the new tracing method and CPU detection schemes speedups of up to 22.53×, with a geometric mean of 2.16× were possible at 32-thread execution and that for some benchmarks the detection schemes continued to provide additional speedups as additional threads were added. The GPU evaluation demonstrated that utilising additional auxiliary devices was a possible technique for offloading some of the processing of speculative execution achieving up to 14.74× with a geometric mean of 1.99× speedup on a 24-thread hyperthreaded machine. Compared to CPU-only reduction-tree scheme this performance is slightly worse with a geometric mean of 0.96× speedup which is likely to improve with upcoming hardware architectures.

This work also showed that removing the version control system was a possible way of achieving higher performance, even if occasional dependence violations occurred. Thanks to the speedups obtainable, all but one of the benchmarks showed it was possible to obtain improved performance even if 2 in every 5 executions failed due to a dependence violation, and in one case 19 out of every 20 executions could fail and speedup would still be achieved in a *kill-and-restart* scenario. For many benchmarks where performance peaked at a lower thread count than was available, some CPU resources could be reassigned to execute the program sequentially at the same time in a *competititive scheduling* scenario resulting in at worst near-sequential execution speeds.

8.1.3 GPU-Based Speculative Execution

Chapter 7 introduced a new speculation scheme that targetted a relatively unexplored platform, the GPU. The scheme's recovery technique mimics that of PARAGON [37] whereby a sequential version of each speculatively parallel section is executed in tandem to the speculative GPU execution. If a dependence is detected or the GPU version takes longer to execute than the sequential version it is killed and the sequential results are used. This technique ensures both correct execution and at worst approximately sequential speeds.

The new scheme improves on PARAGON's speculation design through the use of a faster and more accurate detection scheme. Inspired by an existing CPU-based scheme [30], the new GPU scheme records the highest iteration of the kernel to access a given memory location, triggering a dependence violation if a lower iteration

reads a location written to by a higher iteration, or writes to a location already read by a higher iteration. This technique improves on PARAGON by allowing correctly ordered dependences and also allowing for multiple accesses of the same location. It also performs the checking *on-the-fly* instead of in a separate kernel, improving execution times and reducing the required memory footprint. The new GPU scheme proved effective by providing speedups of up to 99× and on average 3.2× that of sequential. The scheme demonstrated it was possible to outperform standard static auto-parallelisation by a factor of 7× and achieve approximately 99% of the speedup obtained from manual parallel implementations.

8.1.4 Pipelined Speculation Scheme

Chapter 4 presented a new CPU based speculation scheme that provided speedups on par with other existing schemes, performing better in some circumstances. This scheme is intended to complement the existing range of CPU-based speculative schemes allowing for further options and customisation to achieve speedup in speculative scenarios.

8.2 Analysis and Future Work

This section discusses a series of related ideas that has not been investigated by this thesis along with a series of points that could be investigated in future work.

8.2.1 Limitations of Policy Selection

The policy selection techniques investigated and evaluated in Chapter 5 came with several explicit limitations. Most importantly the policy selector required an expensive *off-line* training period for every architecture that uses it. Whilst this process is completely automatable, the expense is undesirable and the results may not take into account existing load on the system. One possible solution to this is to perform the training *on-the-fly* during the execution of the program. Doing so would likely result in additional costs whilst executing, however may also result in more accurate predictions and overall, larger performance increases. It would also further simplify the automa-

tion process. Related to this, the policy selection scheme was also performed *off-line* at compile time resulting in a static selection of speculative loops. Alternatively the policy selections could be performed at runtime enabling some more dynamic aspects, such as machine load, to be considered during selection.

Additionally, one of the more complex tasks not considered by the policy selector is that of memory access patterns, resulting, for instance, in the inability to select appropriate hashing techniques and sizes for speculative policies. This is a complex issue that warrants individual research as the selection of an incorrect hash often results in the detection of *false* data dependences, ultimately resulting in a program that executes slower than the original sequential version.

8.2.2 Scalable Centralised Error Detection

One of the factors preventing further scalability of the error detection scheme evaluated in Chapter 6 is the increasing cost of performing the dependence check at higher thread counts. The reduction tree scheme and abilty to offload to an external computation device aid in this somewhat however for some benchmarks the speedups obtainable still peak at lower thread counts than the number of cores available.

One potential solution to this is to use an alternative, centralised tracing method based on a hybrid version of the page table and SPLIP [30]. An alternative to hashed accesses, potentially causing *false* data dependences, is for the detection scheme to be completely accurate with a page reclaiming process once they become no longer required keeping the memory footprint as small as possible. Such a scheme would ultimately make each speculative access more expensive and raise the synchronisation costs between each thread, however the dependence check would be performed *on-the-fly* and equally distributed between the threads instead of a larger block computation, hopefully resulting in a more scalable scheme.

8.2.3 Block Tracing

One of the factors slowing down speculative execution is the length of time it takes to perform a single speculative access. However, during the evaluation of this work it became clear that many benchmarks perform a set of accesses across sequential addresses. These addresses require tracking code due to the base address of the set being

uncalculatable at compile time, however static analysis can be performed extracting these sequences, and possibly other patterns of memory access, with only the base address being determined at runtime. In these circumstances it may be preferable to collate these sequences into a bulk update of a memory trace, particularly for lazy detection schemes. For instance, in the case of the page table evaluated in Chapter 6 a single access would update a single bit of a page, a slow operation. Instead multiple sequential accesses could be combined into an update more suitable for modern CPU designs, such as performing an update to the page table on a word-level granularity. Doing so would likely substantially increase the speed of performing memory traces in certain circumstances, increasing the overall performance whilst also being orthogonal to existing memory trace mechanisms where single-bit writes are still required.

8.2.4 Combination CPU-GPU Speculation

During the evaluation of the GPU-based scheme in Chapter 7 it became apparent that in many cases the GPU is unsuitable for execution purposes. In some cases, even when the GPU scheme provides a speedup CPU-based schemes may provide a larger speedup. One possible solution to this is to expand on the GPU scheme's protection model to include a speculative version of each loop to execute on the CPU alongside the sequential and GPU versions. This could act as a further form of *competitive scheduling* ensuring only the result of the fastest execution is used whilst also ensuring safety from data dependences.

Bibliography

[1] CoSy compiler framework. http://www.ace.nl/compiler/cosy.html.

[2] NAS parallel benchmarks 2.3, OpenMP C version. http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html.

[3] OpenMP parallel framework. http://openmp.org/wp/openmp-specifications/.

[4] G. Anthes. The power of parallelism. *Computerworld*, November 2001.

[5] E. Bangeman. Intel pulls the plug on 4ghz Pentium 4, Ars Technica, Oct 2004. http://arstechnica.com/uncategorized/2004/10/4311-2/.

[6] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151, Nov 2012.

[7] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPOPP '90, pages 21–30, New York, NY, USA, 1990. ACM.

[8] Philippe Charles and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[9] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 13–24, New York, NY, USA, 2000. ACM.

[10] Marcelo Cintra and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 28:13–24, May 2000.

[11] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[12] D. Grewe, Zheng Wang, and M.F.P. O'Boyle. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *CGO '13*.

[13] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[14] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, Shih-Wei Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, Dec 1996.

[15] P. Hammarlund, A.J. Martinez, A.A. Bajwa, D.L. Hill, E. Hallnor, Hong Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R.B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *Micro, IEEE*, 34(2):6–20, Mar 2014.

[16] Trevor Hastie and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2001.

[17] Lorin Hochstein and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, Washington, DC, USA, 2005.

[18] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Nikolas Ioannou and Marcelo Cintra. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, Washington, DC, USA, 2010.

[20] Brian Jeff. Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1143–1146. ACM, 2012.

[21] Kevin Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *In Proc. ACM/SIGAPP Symp. on Applied Computing*, pages 796–804. ACM Press, 1993.

[22] Leslie Lamport. The parallel execution of DO loops. *Commun. ACM*, 17:83–93, February 1974.

[23] O'Boyle M and Bull M. Expert programmer vs automatic parallelisation: Two approaches to shared virtual memory. *Scientific Programming*, March 1996.

[24] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *SIGPLAN Not.*, 44:166–176, June 2009.

[25] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, New York, NY, USA, 2009.

[26] Ingo Mierswa and Timm Euler. YALE: Rapid prototyping for complex data mining tasks. In Lyle Ungar, Mark Craven, Dimitrios Gunopulos, and Tina Eliassi-Rad, editors, *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, NY, USA, August 2006.

[27] C. Moore. Data processing in exascale-class computer systems. *The Salishan Conference on High Speed Computing*, April 2011.

[28] Cosmin E. Oancea and Alan Mycroft. A lightweight model for software thread-level speculation (TLS). In *Proceedings of the Conference on Parallel Architecture and Compilation Techniques*, PACT '07, Washington, DC, USA, 2007.

[29] Cosmin E. Oancea and Alan Mycroft. Software thread-level speculation: an optimistic library implementation. In *Proceedings of the Workshop on Multicore Software Engineering*, IWMSE '08, New York, NY, USA, 2008.

[30] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, New York, NY, USA, 2009.

[31] Keshav Pingali. Why compilers have failed and what we can do about it. *Keynote Languages and Compilers for Parallel Computing*, 2010.

[32] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 1–12, New York, NY, USA, 2003. ACM.

[33] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[34] Arun Raman and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 65–76, New York, NY, USA, 2010. ACM.

[35] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[36] Peter Rundberg and Per Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *J. Instruction-Level Parallelism*, 3, 2001.

[37] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: Collaborative speculative loop execution on gpu and cpu. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 64–73, New York, NY, USA, 2012. ACM.

[38] Vivek Sarkar. Programming challenges for petascale and multicore parallel systems. In *Proceedings of the 3rd International Conference on High Performance Computing and Communications*, Berlin, Heidelberg, 2007.

[39] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, Jun 1997.

[40] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IISWC '11*.

[41] Byoungro So, Sungdo Moon, and Mary W. Hall. Measuring the effectiveness of automatic parallelization in SUIF. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 212–219, New York, NY, USA, 1998. ACM.

[42] Gurindar Sohi. Rethinking parallel execution for future multicore processors, 2011.

[43] J. Gregory Steffan. *Hardware support for thread-level speculation*. PhD thesis, School of Computer Science, Pittsburgh, PA, USA, 2003. AAI3159472.

[44] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–187, New York, NY, USA, 2009. ACM.

[45] Khai Q. Tran, Spyros Blanas, and Jeffrey F. Naughton. On transactional memory, spinlocks, and database transactions. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010, Singapore, September 13, 2010*, pages 43–50, 2010.

[46] Ilkka Tuomi. The Lives and Death of Moore's Law. *First Monday*, 7(11), 2002.

[47] Rajeshwar Vanka and James Tuck. Efficient and accurate data dependence profiling using software signatures. In *CGO '12*.

[48] Tobias J. K. Edler von Koch and Björn Franke. Variability of data dependences and control flow. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 180–189, 2014.